Approved for public release; distribution is unlimited.

Broadband Modal Beamforming of Acoustic Tomography Signals Acquired by a Vertical Array

by

Glenn A. Omans II
Lieutenant, United States Navy
B.S., Virginia Military Institute

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN APPLIED SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

September 1992

ABSTRACT

The objective of this thesis is to develop a technique and associated algorithms to extract the arrival time of modal energy, using a vertical array, from broadband signals. Modal energy arrival time is important to shallow water acoustic tomography because low angle rays, which contain the majority of acoustic energy, are often not resolvable. Tilt compensation is included in the beamforming algorithm to provide a virtual vertical array. A broadband modal filtering technique is accomplished through weighting the frequency components of phase encoded tomographic signals by the spectrum of the mode shapes. A methodology of phase decoding after beamforming was adopted to minimize processing. Initial development and prototyping was done using a parabolic equation model. Further testing was accomplished on real data taken from the Barents Sea Polar Front Experiment, August 1992. Results show consistency over a number of transmitted pulses. Mode energy travel time measurement is simplified due to the distinct arrival structure of beamformed signals. Based on these results, the modal beamforming algorithm should be a useful tool for acoustic tomograpy.

ified

CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

RT SECURITY CLASSIFICATION JNCLASSIFIED	1b RESTRICTIVE MARKINGS
ITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
ASSIFICATION / DOWNGRADING SCHEDULE	
RMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
OF PERFORMING ORGANIZATION al Postgraduate School	6b OFFICE SYMBOL (If applicable) 7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
ESS (City, State, and ZIP Code)	7b ADDRESS (City, State, and ZIP Code)
OF FUNDING / SPONSORING NIZATION	8b OFFICE SYMBOL (If applicable) 9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
ESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

(Include Security Classification) BROADBAND MODAL BEAMFORMING OF ACOUSTIC TOMOGRAPHY SIGNALS ACQUIRED BY A VERTICAL ARRAY

NAL AUTHOR(S)
nn A. Omans II

OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
ERS THESIS	FROM _____ TO _____	September 1992	120

EMENTARY NOTATION
THE VIEWS EXPRESSED IN THIS THESIS ARE THOSE OF THE AUTHOR AND DO NOT REFLECT THE OFFICIAL POLICY OR POSITION OF THE DEPARTMENT OF DEFENSE OR THE U.S. GOVERNMENT.

COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
	GROUP	SUB-GROUP	ACOUSTIC TOMOGRAPHY, BROADBAND MODAL BEAMFORMING, PHASE ENCODED SIGNALS

ACT (Continue on reverse if necessary and identify by block number)

jective of this thesis is to develop a technique and associated algorithms to t the arrival time of modal energy, using a vertical array, from broad band signals energy arrival time is important to shallow water acoustic tomography because low rays, which contain the majority of acoustic energy, are often not resolvable. ompensation is included in the beamforming algorithm to provide a virtual vertical

A broadband modal filtering technique is accomplised through weighting the ncy components of phase encoded tomographic signals by the spectrum of the mode . A methodology of phase decoding after beamforming was adopted to minimize pro- g. Initial development and prototyping was done using a parabolic equation model. r testing was accomplished on real data taken from the Barents Sea Polar Front ment, August 1992. Results show consistency over a number of transmitted pulses. nergy travel time measurement is simplified due to the distinct arrival structure unformed signals. Based on these results, the modal beamforming algorithm should

BUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION	
LASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS		
E OF RESPONSIBLE INDIVIDUAL Miller	22b TELEPHONE (Include Area Code) (408) 646-2384	22c OFFICE SYMBOL EC/Mr

1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

S/N 0102-LF-014-6603 Unclassified

be a useful tool for acoustic tomography.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

This work would not have been possible without the support of my family, many dear friends and colleagues. I pay tribute to my parents, who are responsible for making me look inward to find the answers to life's mysteries. Thank you mother for teaching me to believe in myself, and thank you father for giving me your support. To the rest of my family, I give my sincerest appreciation for always being there with love and understanding.

A special thank you to my advisor Jim Miller who inspired me, placated my ego and occasionally zapped me with his acoustic cattle prod when I started to slow down. My sincerest appreciation goes out to my co-advisor Joshua Rovero and second reader Ching-Sang Chiu for always answering my sometimes misguided questions. I owe thanks to Jim Lynch, Arthur Newhall, Rich Palowitz and the rest of the Woods Hole Gang for taking me in to their select group and baptizing me into the world of acoustic tomography.

For my loving wife, I know this process has been as much a test for you as it has for me. Rebecca, thanks for standing by me.

I. INTRODUCTION

A. BACKGROUND

Tomography signal processing requires the precise measurement of acoustic travel time, which is the integral along the raypath of inverse sound speed. Travel time of acoustic signals has been well determined to be a function of temperature, salinity, and pressure. Once time perturbations for multiple arrival paths are known, ocean fluctuations can be determined using mathematical inverse techniques [Ref. 1]. In addition to measuring time perturbations based on ray arrival times, it is possible to measure perturbations based on mode arrivals. There are two approaches to tomography using modes: for continuous wave signals, the basic measurement is the modal phase difference; for broad band signals, the measurement used is modal travel time [Ref. 2]. Since coded signals (maximal-length sequence phase encoded signals) are intrinsic broadband signals, modal travel time measurement is germane to the work done in this paper. Where ray theory is not applicable, or not convenient, it is possible to extend tomographic techniques using both rays and modes [Ref. 3].

The goal of my thesis is to develop a technique for broadband modal beamforming for acoustic tomography. The

scope entails broadband modal beamforming with a vertical array and decoding (m-sequence removal of phase encoded signals). In the process of my research several software programs were developed. Most notable is the broadband modal beamformer, which is an adaptation of the continuous wave beamformer developed by Crocker [Ref. 4]. Specific goals of research include:

- Develop a broad spectrum modal filtering technique given the normal mode's eigen function and values.
- Evaluate compatibility for phase sequence removal prior to and following beamforming.
- Minimize processing time, where possible, and provide for a robust environment.
- Evaluate performance using both synthetic and in situ data.
- Provide a virtual array based on, time varying array tilt, array geometry and modal group speed.
- Incorporate a supporting cast of programs, including a modal decomposition program for standardizing input to the beamformer and to simplify and enhance future signal processing.

B. THESIS SUMMARY

Chapter II, **Barents Sea Experiment**, is a summary of the Barents Sea Polar Front Experiment (BSPFEX). This experiment is addressed because it provided real data to evaluate the processing algorithms.

Chapter III, **Acoustic Normal Mode Propagation Theory**, is an introduction to normal modes. Modal acoustic propagation

is addressed. The chapter is concluded with a section linking acoustic ray and normal mode theory.

Chapter IV, **Signal Processing Concepts**, presents the concepts and theory behind broadband modal beamforming. Subjects include beam steering, mode filtering and phase encoded signals. A plane wave beam steering approach is taken to compensate for array tilt. Following tilt correction, my modal filtering methodology is presented. Finally, a brief introduction to m-sequence signal decoding is given. Phase preservation after beamforming is addressed, supporting the concept of decoding after beamforming.

Chapter V, **Barents Sea Array**, is a description of the array used during the Barents Sea Polar Front Experiment (BSPFEX). Improvements made from previous vertical arrays are discussed. Statistics on hydrophone performance as well as the array modal beam patterns are addressed.

Chapter VI, **Evaluation of Barents Sea Data**, is a presentation of results from the BSPFEX. Comparisons and conclusions are drawn concerning the array's performance.

Chapter VII, **Summary and Future Work**, presents a summation of conclusions and identifies areas requiring improvement.

The appendix contains the most significant algorithms I developed in the course of research. Algorithms included are: the broad band modal beamformer, modal decomposition program and the supporting modal extraction program which formats the input files for use in the beamformer.

II. BARENTS SEA EXPERIMENT

Concepts developed in this thesis were applied to real data collected from the Barents Sea Polar Front Experiment (BSPFEX). During the BSPFEX, I developed and tested my beamforming techniques and algorithms on synthetic data modeled using the *Finite Element Parabolic Equation (PE) Model* developed by Collins and Westwood [Ref. 5]. The synthetic data was modeled for the anticipated location of the array for the BSPFEX. Unfortunately, the array was deployed in a different location from the model limiting any comparisons drawn between the synthetic and BSPFEX data.

The BSPFEX was conducted during August of 1992. The experiment comprised a joint effort between the Naval Postgraduate School (NPS), Woods Hole Oceanographic Institution (WHOI), and the Science Applications International Corporation (SAIC). The principal investigators for the experiment are Professors Robert Bourke, Ching-Sang Chiu, and James H. Miller from NPS, Dr. James F. Lynch and Dr. Al J. Plueddemann from WHOI, and Dr. Robin Muench from SAIC. The principal engineer for the vertical hydrophone array system was Mr. Keith Von der Heydt from WHOI. The objectives of the experiment as outlined by the Barents Sea Polar Front Group 1992 are:

1. Provide a detailed physical description of the polar front.

2. Enhance the understanding of dynamics of the front, including frontogenesis and its influence on regional oceanographic processes.

3. Assess the ability of acoustic tomography to define frontal and associated mesoscale features.

4. Provide improved documentation of shallow water acoustic propagation in this region and the effect of the environment on acoustic ASW operations.

[Ref. 6]

This experiment is a milestone in acoustic tomography. It is the first time that a vertical array has been used in a shallow water environment to conduct acoustic tomography. A sixteen hydrophone telemetered vertical array, with ten meter spacing between hydrophones, was deployed during the experiment. The source and array mooring locations are listed in Table I. Table II describes the characteristics of the deployed broadband sources.

Table I MOORING LOCATIONS FOR THE 224 AND 400 HZ SOURCES AND THE VERTICAL ARRAY.

Mooring	Latitude	Longitude	Depth (m)
SE(a) VLA	74° 19.1512'N	23° 33.1438'E	275.0
SE(b) VLA	74° 19.1996'N	23° 32.2960'E	275.0
NE 224 Hz	74° 37.5535'N	23° 24.3755'E	142.0
NW 400 Hz	74° 32.9152'N	21° 44.1043'E	176.0
SW 400 Hz	74° 04.7337'N	22° 00.4605'E	380.0

Table II BSPFEX SOURCE CHARACTERISTICS.

Mooring	**NE**	**SW**	**NW**
Freq (Hz)	224	400	400
SL (dB)	183	183	183
Duration (s)	118.25	132.86	132.86
BW (Hz)	16	100	100
Q (cyc/dig)	14	4	4
No. digits	63	511	511
Seq length	3.9375	5.11	5.11
No. sequences	30	24 +2	24 +2
Cycle (min)	2.5,7.5 ...	0,10,20 ...	5,15,25 ...

Overall, the experiment was a sterling success. The WHOI-NPS vertical array was deployed on 12 August 1992. The array performed flawlessly for fourteen hours when a failure in the amplifier chip forced retrieval and repair of the array. After repair, the array was redeployed without the adjustable gain control making signal quantization from analogue to digital less optimal. Following recovery and redeployment, the array operated continuously for three days recording signals from both the 224 Hz and 400 Hz sources. In all, fifteen gigabytes of data were recorded.

One major disappointment was the failure of the NW 400 Hz transceiver to transmit shortly after its deployment (see Figure 1). No attempt was made to recover and repair the broken transceiver so as to not disturb the tomographic deployment schedule, and in the hopes that the transceiver was still able to receive acoustic data. Post experiment recovery of the broken transceiver revealed that it was still able to

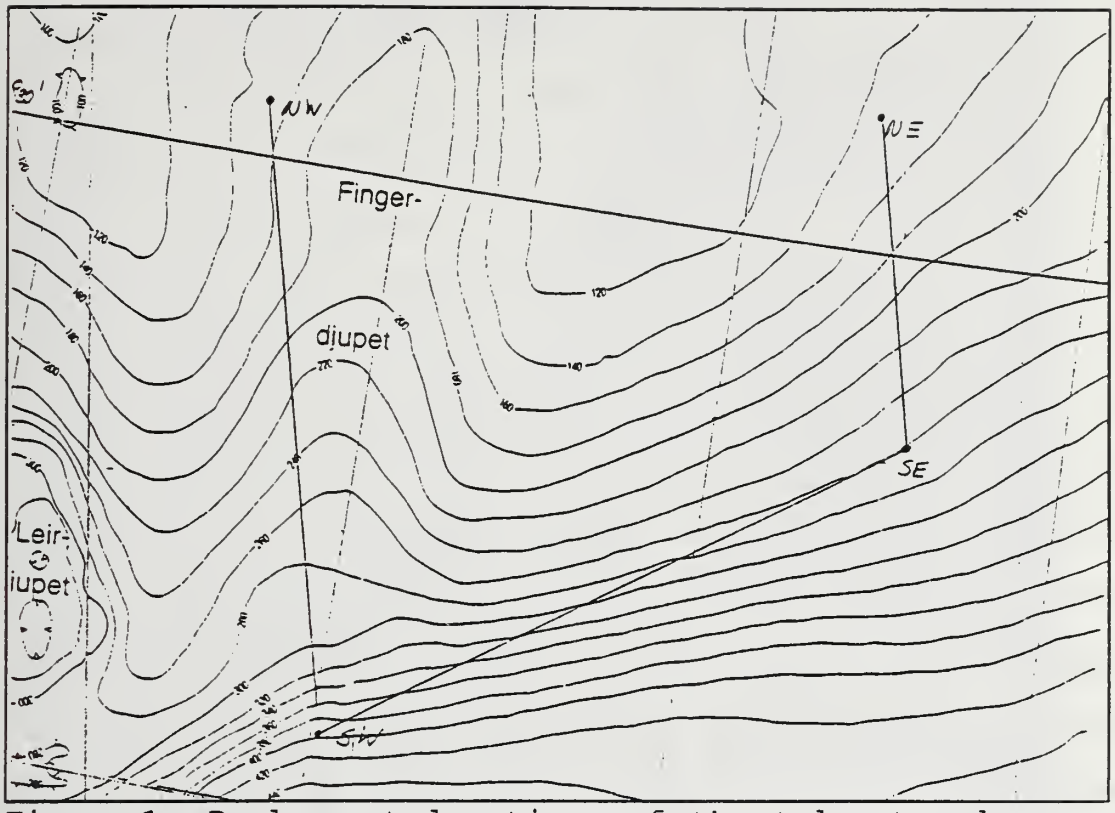

Figure 1 Deployment locations of the telemetered array and sources, from the BSPFEX. SW - 224 Hz transceiver; SE - 16 channel vertical array; NW/NE - 400 Hz transceivers [Ref. 7].

receive and recorded three days of acoustic data. With the NW transceiver still partially operational, three paths for tomography are available; two across the polar front and one along the front.

In addition to tomographic pulse receptions, other acoustic observations were conducted. The ship dropped eight SUS charges to measure reverberation and bottom properties. Eighteen additional SUS charges were dropped from a P-3 aircraft along the southwestern and northern tracks, to be used in evaluating propagation and bottom loss. One hour of continuous wave (CW) 224 Hz transmission was made for use in

modal phase and matched field tomography, and to measure temporal coherence for comparison with pulsed transmissions.

Other sensors used during the deployment include, over three hundred Conductivity-Temperature-Depth (CTD) deployments, two of three current meters (one failure), and acoustic doppler current profiler (ADCP). Figure 2 denotes the fine structure of the polar front present, as measured from CTD deployments. Acoustic propagation was excellent during the experiment. The 224 Hz and 400 Hz sources were clearly audible at ranges over 50 kilometers, even without signal processing. With such clear receptions and low noise levels, the task of processing the data later will be much simplified. The above summary was based on the preliminary cruise report for the BSPFEX [Ref. 7].

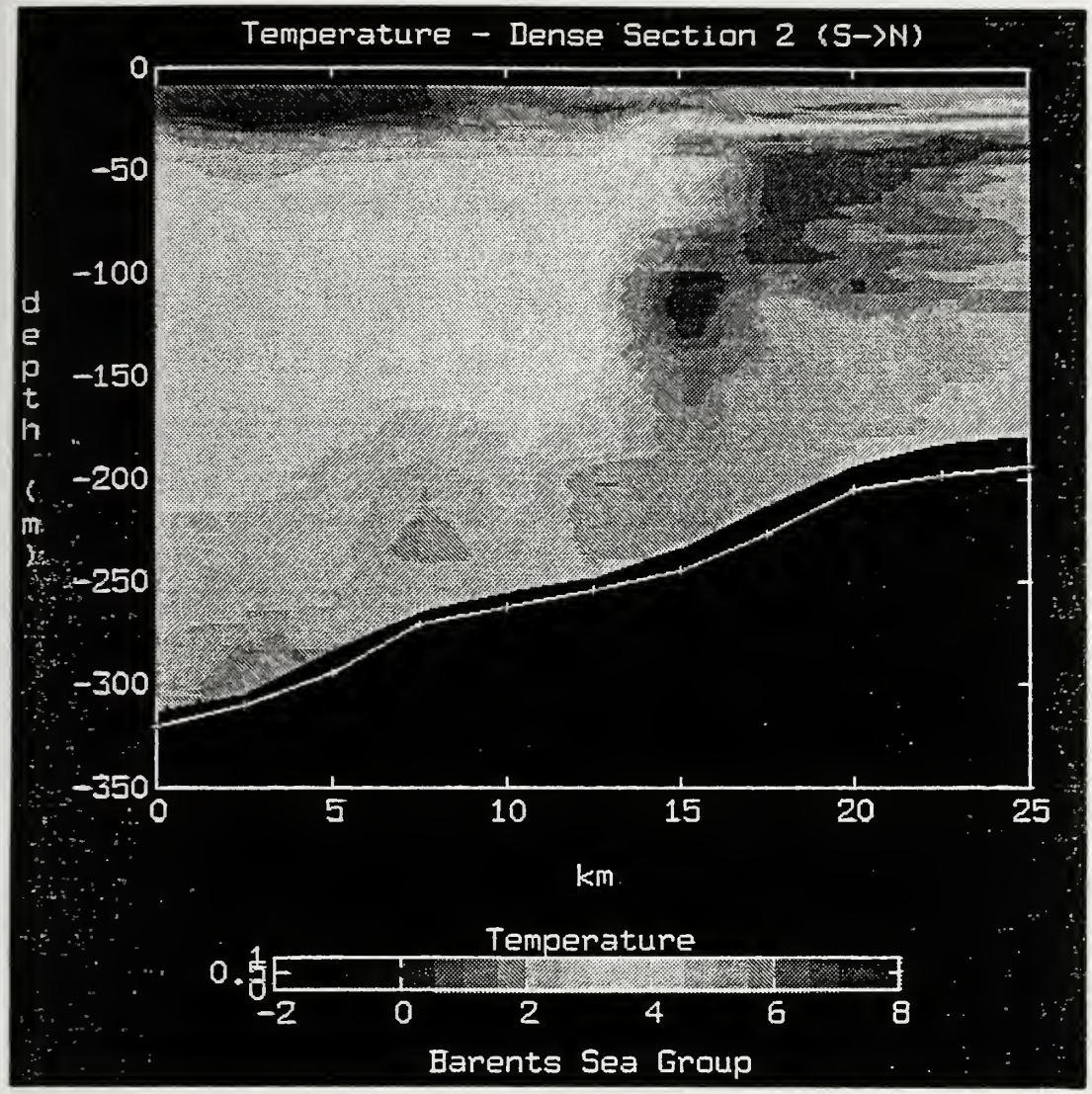

Figure 2 Contour plot of the Barents Sea Polar Front [Ref. 7].

III. NORMAL MODE PROPAGATION THEORY

A. ADIABATIC MODE THEORY

Normal mode propagation theory is a physical model whose eigenfunctions, are solutions to the acoustic wave equation,

$$\nabla^2 p = \frac{1}{c^2}\frac{\partial^2 p}{\partial t^2}. \tag{3.1}$$

in a waveguide [Ref. 8]. The objective of normal mode theory is to model the "local" pressure field as a linear combination of vertical standing waves. The term local refers to the range dependent sound speed structure and boundary conditions at a certain range. The following development applies for a slowly varying range dependent waveguide and is taken from Shang [Ref. 2].

The acoustic pressure field can be expressed in terms of local mode superposition,

$$p(r,z) = \sum_{n=1}^{\infty} \frac{F_n(r)\,\psi_n(z,r)}{\sqrt{r}}, \tag{3.2}$$

where $\Psi_n(z,r)$ is the local solution or eigenfunction, and describes the distribution of modal energy as a function of depth. $F_n(r)$ is the range dependent portion of the pressure field, and the $\sqrt{r}$ term describes the geometric transmission losses caused by cylindrical spreading.

Subject to the boundary conditions and orthogonality, the local mode Ψ_n must satisfy,

$$\frac{\partial}{\partial z}\left(\frac{1}{\rho(z)}\frac{\partial \psi_n}{\partial z}\right) + \frac{1}{\rho(z)}\left(k^2(r,z) - k_n^2(r)\right)\psi_n = 0, \qquad (3.3)$$

where $k^2(r,z)$ is the acoustic wave number, $k_n^2(r)$ is the vertical wave number, or the eigen value, associated with the mode Ψ_n. $\rho(z)$ is the depth dependent density. The range dependent portion of the pressure field must satisfy,

$$\frac{d^2F_m}{dr^2} + \left(k_m(r) + \frac{1}{4r^2}\right)F_m = -\sum_n \left[2A_{mn}(r) + B_{mn}(r)\,F_n\right]\frac{dF_n}{dr}. \qquad (3.4)$$

A_{mn} and B_{mn} are coefficients describing the first and second order coupling effects in the sound channel:

$$A_{mn} = \int \frac{1}{\rho_o(z)}\psi_m(z,r)\frac{\partial \psi_n(z,r)}{\partial r}dz, \qquad (3.5)$$

$$B_{mn} = \int \frac{1}{\rho_o(z)}\psi_m(z,r)\frac{\partial^2 \psi_n(z,r)}{\partial r^2}dz. \qquad (3.6)$$

If mode coupling is weak enough, then coupling can be neglected, and we have the adiabatic solution,

$$p(r,z) = e^{j\pi/4}\sqrt{2\pi}\sum_m \frac{\psi_m(z|0)\,\psi_m(z|r)}{\sqrt{k_n r}} \times e^{\left(j\int_0^r k_m(R')\,dR'\right)}. \qquad (3.7)$$

B. NORMAL MODES AND ACOUSTIC TOMOGRAPHY

The travel time of each normal mode is dependent on the ocean structure. It is possible to estimate mode travel time from acoustic models, or it can be directly measured using mode filtering techniques. The difference between measured and estimated arrival times are the time perturbations for each mode. Broadband modal acoustic tomography seeks to exploit these time perturbations by applying inverse techniques to infer information about the acoustic environment.

For a broadband source, the pressure field can be described as a function of a continuous wave field modified by the source signal spectrum,

$$p(r,z,t) = \int_{-\infty}^{\infty} s(\omega)\, p_{cw}(r,z,\omega)\, e^{-j\omega t} d\omega, \qquad (3.8)$$

where $s(\omega)$ is the signal spectrum of the source, and p_{cw} is the continuous wave field. The total pressure is the sum of the pressures for every mode,

$$p(r,z,t) = \sum_{m} p_m(r,z,t), \qquad (3.9)$$

where the modal pressure term is,

$$p_m(r,z,t) = \int_{-\infty}^{\infty} s(\omega)\, \frac{\psi_m(z_o)\,\psi_m(z)}{\sqrt{\kappa_m r}}\, e^{j(\phi_m(r)-\omega t)} d\omega. \qquad (3.10)$$

Taking the derivative of the phase with respect to frequency,

$$t_m \cong \frac{d\kappa_m}{d\omega} r, \tag{3.11}$$

and it follows that the group speed must be,

$$c_m \cong \frac{d\omega}{d\kappa_m}. \tag{3.12}$$

It has been shown that there is a unique travel time associated with each mode for a given environment [Ref. 2]. Acoustic tomography can be accomplished using the time perturbations taken from the measured and estimated arrival times. The remaining task is to develop a technique to measure the modal arrival times, which is the subject of the following chapter on signal processing.

IV. SIGNAL PROCESSING CONCEPTS

A. BROAD BAND MODAL BEAMFORMING

Wave propagation modeling can be used to extract temporal and spatial characteristics for a propagating wave, and through inverse techniques to infer information about the acoustic medium. This is the essence of acoustic tomography. In recent years, ocean modeling for acoustic tomography has been dominated by acoustic ray theory. Ray theory is attractive because it is simple to model and requires minimal computer resources. Other modeling techniques simply were not practical until recent years. Ray theory is a high frequency approximation which fails to describe turning points, diffraction leakage and caustics. Further, many ray arrivals may not be resolvable, particularly in a shallow water environment.

A vertical array provides an added dimension to tomography. Not only does it increase signal gain, it also allows mode and plane wave beamforming as well as enhancing resolvability of ray arrivals. The rest of this section will be devoted to the concepts required for modal beamforming.

1. Beam Steering

Array tilt can be caused by a number of forces such as waves, ocean currents, and external tensioning. Any tilt in the array is undesirable, and must be corrected. A virtual vertical array can be established using a plane wave beam steering methodology.

There are essentially two approaches to plane wave beam steering. The first applies phase shift delays to the signal at the desired frequencies. The second uses true time delays. Phase delay beam steering is well suited for CW signals. However, this method is much less efficient for broad band signals because it requires phase shifting for every frequency of the broad band signal.

Time delay beam steering is more suitable for use with broadband signals. Time shift delays are determined from array geometry and sound speed. For a line array, the time shift delay for the n_{th} element can be described as,

$$\tau_n = \frac{nd\sin(\theta_o)}{c}, \qquad (4.1)$$

where θ_o is the tilt in the direction of the source, and d is the element spacing [Ref. 9]. For normal mode beamforming, c is the group speed, equation (3.16), for each mode. Equation (4.1) is elementary in nature, and can be applied to any array or signal. Applying shift delays to discrete time signals

requires interpolation incurring additional processing [Ref. 4].

2. Mode Filtering

Plane wave beamforming does not account for the non-uniform energy distribution in the wave guide as described by normal modes. There are two types of modal beamforming, continuous wave (CW) and broad band (BB).

For CW modal beamforming, the task is simple. The time domain pressure field need only be weighted by the eigenfunction corresponding to each element depth. This method is not appropriate for broad band signals, because normal modes are a function of both depth and frequency. For BB beamforming, the frequency domain signal must be weighted by the corresponding frequency component of the eigenfunction. This requires that the mode shape be known at each frequency of the Discrete Fourier Transform (DFT) for the pressure field covering the desired bandwidth.

For encoded signals with long duration sequence lengths, modal decomposition processing can be unsurmountable. The nominal modal decomposition time is thirteen minutes for each frequency. A 100 Hz bandwidth source with a sequence length of 5.11 seconds has 1022 frequency components. With current computer resources (SPARC 2), it would take over nine days to process the eigen functions for each frequency. Fortunately, the mode shapes are continuous functions in depth and

frequency. Figure 3 shows how the mode amplitude, at a fixed depth, varies slowly with frequency. The mode shapes can be sampled as a function of frequency and then interpolated to achieve and desired frequency increment within the band.

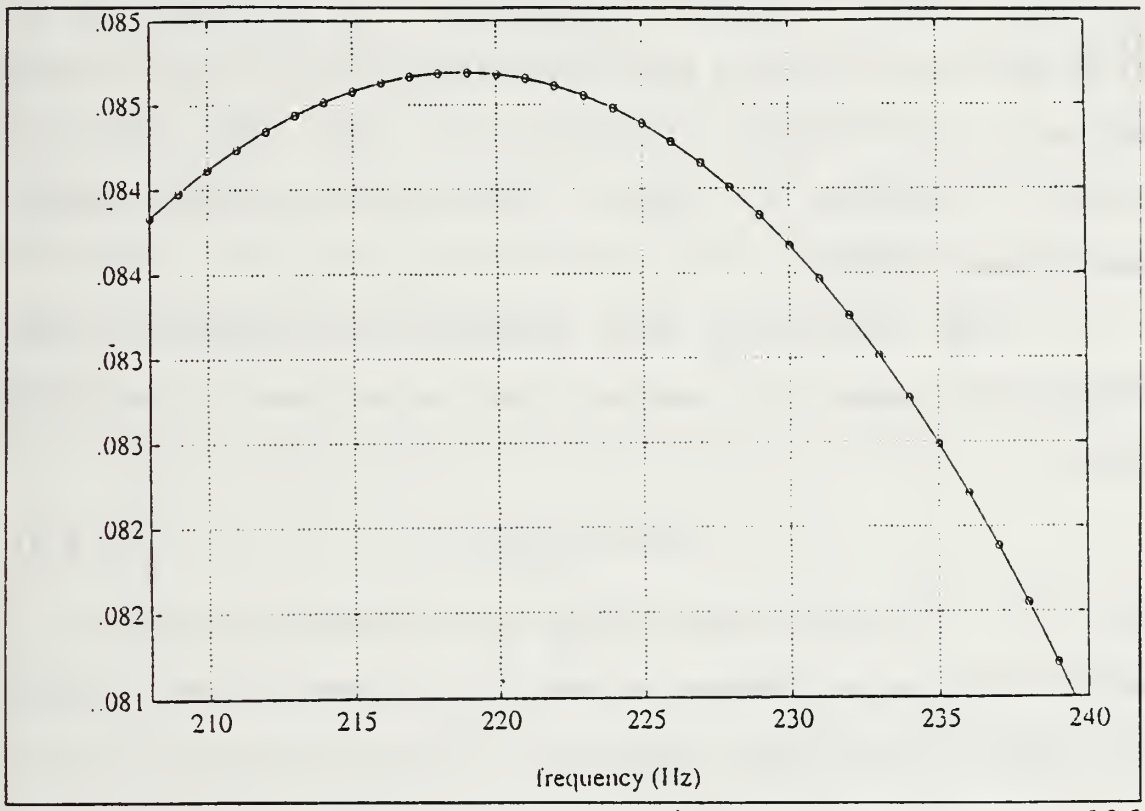

Figure 3 Mode twenty as a function of frequency at 126 meter fixed depth.

The next step is to weight the frequency domain pressure field by the mode shape at depths corresponding to the array elements. The following equation is the essence of modal filtering.

$$a_m(t) = \mathcal{F}^{-1}\left[\sum_n \rho(z)\, P(z_n, f)\, Z_m(z_n, f)\right]. \qquad (4.3)$$

where a_m^2 is proportional to the intensity of mode m, and ρ is the density in the water column. The only limitation of equation (4.2) is that the Fourier Transforms are long enough to accommodate the length of the entire coded sequence. The pressure field frequency components are weighted by the frequency domain normal mode at the appropriate array element depths. The resultant is summed over depth, and an inverse Fourier Transform is applied to achieve the time domain beamformed signal.

The array gain (AG) exhibited by broadband modal beamforming cannot be greater then conventional beamforming theory,

$$AG = 10\log(N), \tag{4.3}$$

[Ref. 10]. The theoretical array gain shown in equation (4.3) assumes the noise between elements is uncorrelated. Signal enhancement is strongly dependent on this correlation factor.

B. MAXIMUM LENGTH PHASE ENCODED SIGNALS

The objective of signal processing for broad spectrum acoustic tomography is the measurement of travel time for different raypaths and mode arrivals along a vertical cross section of the ocean.

One method in which travel time can be measured is through the use of explosive and implosive devices. Unfortunately, explosive devices do not propagate repeatable acoustic signals. A better method that has been found employs the use of maximal-length sequences (m-sequences) or pseudo-random noise are well suited for this

application because of their deterministic nature, correlation properties, and simplicity [Ref. 11].

M-sequences are a sequence of binary bits having the property of a maximum possible period for an r-staged shift register. Shift registers are created from primitive polynomials. These binary shift registers are mapped to a series of ones and minus ones [Ref. 12]. The primary characteristic of the maximum length shift register is its autocorrelation properties. The self correlation of an m-sequence of length N produces a triangular function of short duration as shown at Figure 4.

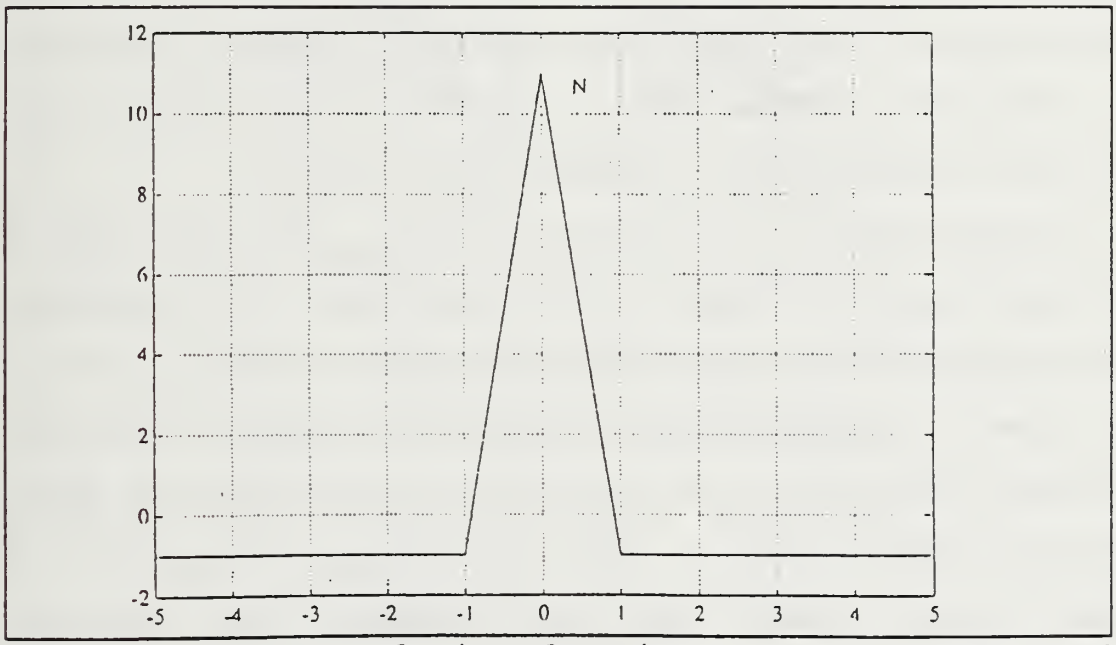

Figure 4 Autocorrelation function for a m-sequence of length N.

An m-sequence coded signal is formed using a simple harmonic source and applying a phase shift from the m-sequence register. A phase encoded signal has the form,

$$s(t) = A\cos(2\pi f_c t + M(t)\phi), \qquad (4.4)$$

where A is the amplitude of the transmitted signal and M(t) is the value of the shift register for a specified digit duration. Q is an integer number of cycles (at carrier frequency) per digit and is used to determine the digit duration. The phase modulation angle Φ is chosen to optimize signal-to-noise performance:

$$\phi = \tan^{-1}(\sqrt{N}). \qquad (4.5)$$

The one major drawback to using coded signals is that the standard correlation requires N^2 multiplications. Decoding algorithms can over come this processing issue by using the *Fast Hadamard Transform* (FHT). The FHT is analogous to a FFT and reduces processing to **NlogN** multiplications.

Decoding signals is a simple process. The first step is to filter the input signal using a band pass filter covering the frequencies containing the coded signal. After filtering the signal is demodulated to remove the carrier frequency. The demodulated signal is then correlated for different shift versions using the FHT to reduce processing. Figure 5 is a block diagram showing the Birdsall-Metzger (BM) decoding process used in [Ref. 11].

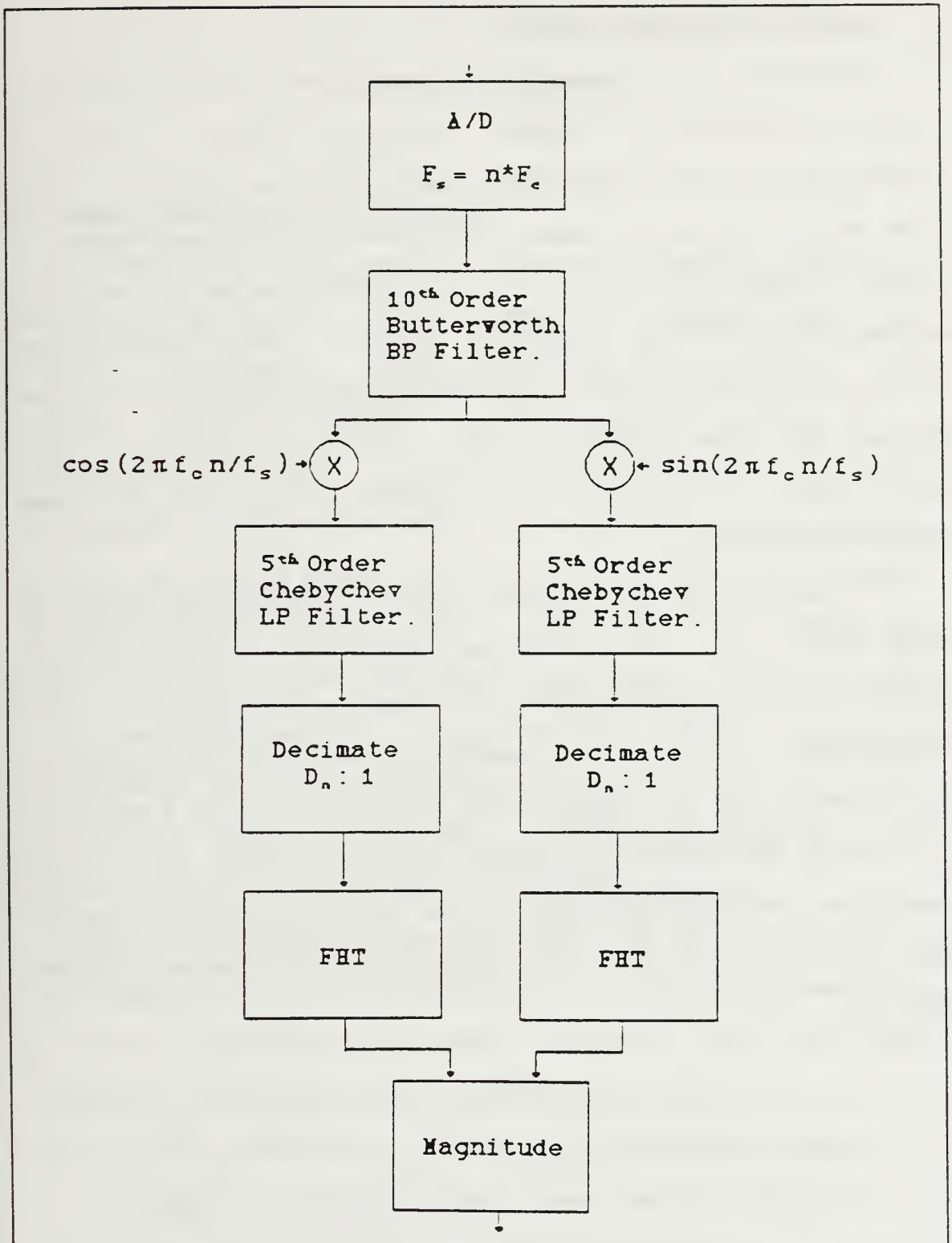

Figure 5 Block diagram of m-sequence removal algorithm [Ref. 11]

C. SIGNAL PROCESSING SUMMARY

The first step in modal beamforming is the correction of tilt to provide a virtual vertical array. This is accomplished using plane wave beamforming techniques. True time shift delays are used vice phase shift beamforming to reduce processing. After tilt correction, mode filtering is accomplished in the frequency domain. Modal beamforming constitutes filtering the frequency domain signals by the mode shapes, and then summing across the array elements. Post beamforming, the signals are decoded using the BM phase decoder algorithm.

The phase decoding process can be accomplished prior to beamforming, but this would significantly increase signal processing as a multiple of the array elements and is not recommended. Further, the beamforming algorithm developed in this thesis does not support this procedure. If at some point it becomes desirable to process signals in this manner, the beamformer program would have to be altered to except complex data and to perform mode filtering about zero frequency offset signals.

V. BARENTS SEA ARRAY

A. ARRAY COMPONENTS AND DESIGN

The WHOI-NPS array contains sixteen hydrophone elements at ten meter intervals. The telemetered vertical array was moored to the bottom using a 1000 lb. cast anchor. The distance from the anchor to the bottom hydrophone is approximately 3 meters. A forty-eight inch bouyant syntactic foam sphere suspends the array 166 meters above its mooring. One meter below the sphere are the dual Benthos Interrogators. The interrogators were used to acoustically determine the exact location of the tethered array and to measure array tilt. Two inclinometers, one located two meters beneath the interrogator and the second near the center of the array, provided an additional means of determining array tilt. Figure 6 is an illustration of the array design.

Nominal hydrophone sensitivity was -160 dB re 1 v/μPa with a low frequency roll-off at 50 Hz. Data was preprocessed with an 8-pole *Butterworth* low pass filter at 500 Hz cutoff frequency. A sampling rate of 1600 Hz was selected to achieve twice Nyquist frequency of the highest frequency source 400 Hz. A sampling frequency four times carrier frequency is required by the m-sequence removal algorithm to simplify

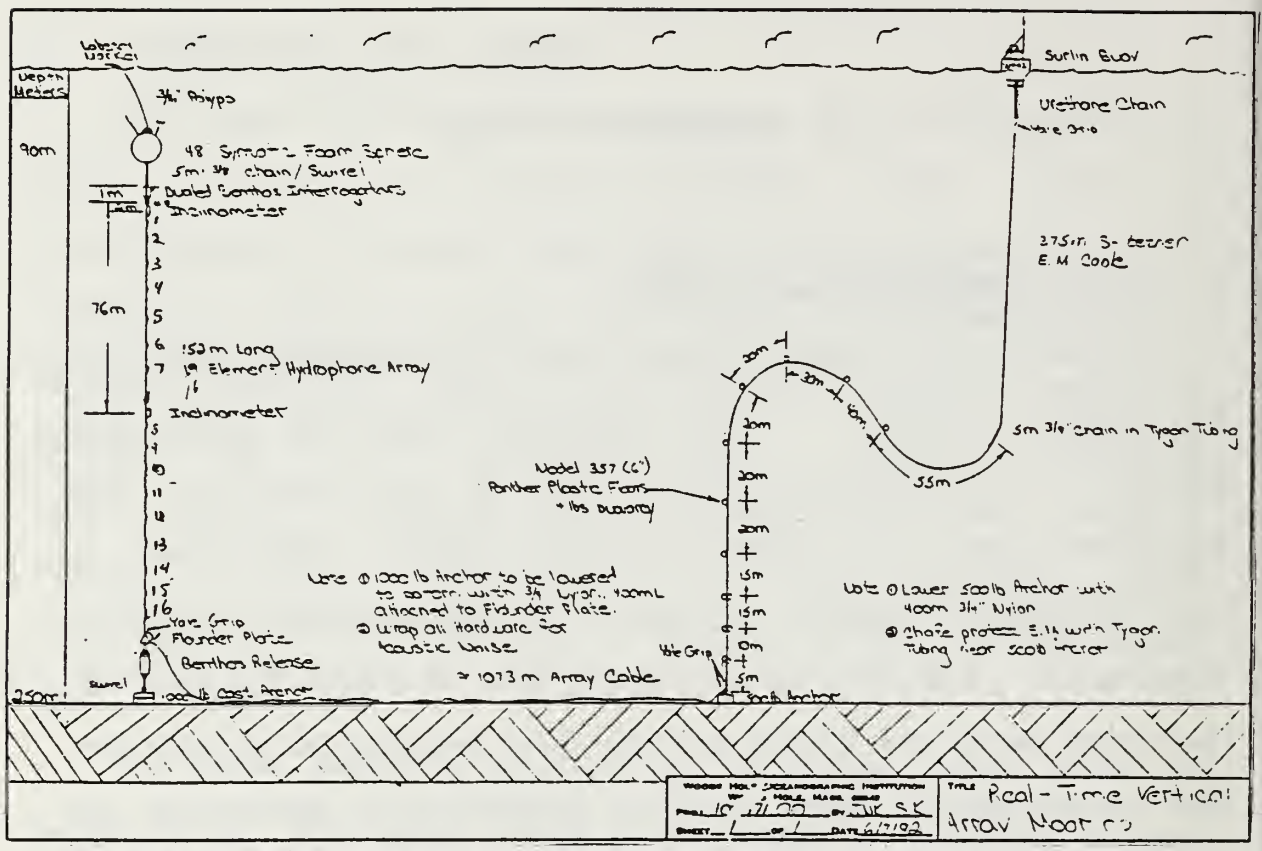

Figure 6 Diagram of the WHOI-NPS vertical array used during the Barents Sea Polar Front Experiment [Ref. 7].

signal processing. Data from the 224 Hz source had to be resampled to 896 Hz sampling frequency for decoding. A single data bus was used to sample all sixteen array channels and created a 30 microsecond skew between elements. The beamformer corrects for the sampling skew based on the principles established for tilt compensation.

B. BARENTS SEA POLAR FRONT EXPERIMENT - MODAL BEAMFORMER

1. Equipment Performance

The WHOI-NPS array was deployed in the Barents Sea on 12 August 1992. The array performed well with one minor

mishap when the amplifier chip failed. The amplifier chip failure effected adjustable gain control and degraded analogue to digital quantization for much of the data collected. All array elements operated without failure, and the array remained operational throughout its deployment.

A new mooring and tether design gave the array increased stability. The mean measured array tilt was three tenths of a degree. This is meaningful considering that previous vertical arrays suffered tilt in excess of ten degrees thereby degrading system performance.

2. Modal Decomposition

Mode filtering performance is highly dependent on array geometry and the normal mode shapes. Modal decomposition requires knowledge of upper and lower boundary conditions, sound velocity, and density. The algorithm used for modal decomposition is located at the appendix of this thesis. [Ref. 13] The decomposition algorithm employs a finite differencing routine to approximate the initial eigen value problem. Implicit to the routine is the assumptions of a pressure release surface and a perfectly rigid lower boundary. The bottom is modeled using constant sound speed and density characteristics. There is no limitation to the quantity or complexity of bottoms except for processing time, which increases as the square of the number of points in the profile.

Sound and density profiles used to calculate eigenvalues and shapes for the Barents Sea were determined from CTD data, and estimations of bottom sound speed and density. All processing is based on a single bottom model with a constant sound speed of 1667 m/s, and density of 1.83 g/cm^3, taken from Clay and Medwin [Ref. 14].

Figure 6 is the Barents Sea sound velocity profile at the location of the array. The sound velocity indicates the existence of a well defined mixed layer followed by a strong negative sound speed gradient extending to the bottom. Due to the negative sound speed gradient, the majority of acoustic energy is expected to be trapped well below the surface with significant bottom interaction.

3. Normal Modes at the Vertical Array

Figures 7 to 12 are the mode shapes for the Barents Sea array at the 224 Hz source center frequency. Modes one through four have turning depths well beneath the surface and are expected to contain the majority of acoustic energy based on the strong negative sound speed gradient. Modes five through ten also turn beneath the surface, but should have less beamformed energy than the first four modes because there is significant modal energy outside the array's aperture. Modes ten through twenty interact with the surface. These modes should have much less energy due to surface scattering effects.

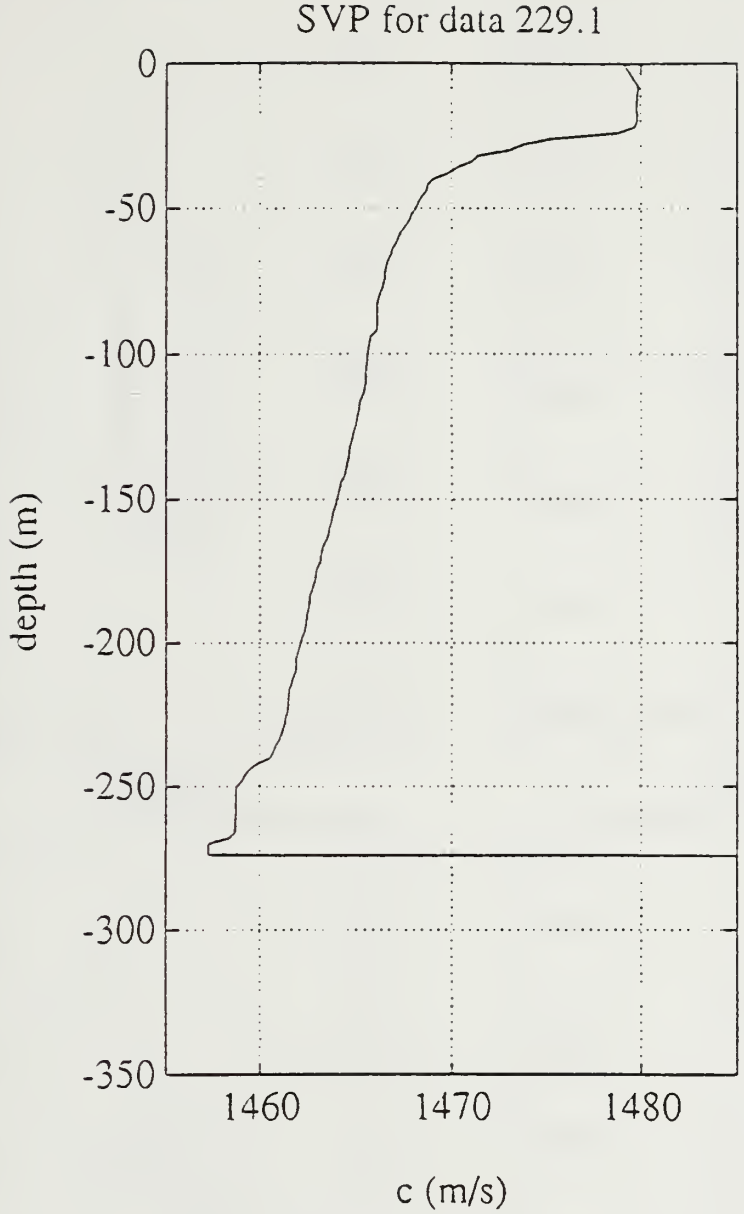

Figure 7 Sound speed profile at the receiving array. Bottom sediment starts at 275 meters modeled by constant ρ 1.83 g/cm^3, and sound speed 1677 m/s; The bottom is assumed rigid below 420 meters.

4. Modal Beam Pattern

There are many measurements of performance for classical arrays. Plane wave beamforming is a well understood process, and performance of a plane wave beamformer can be

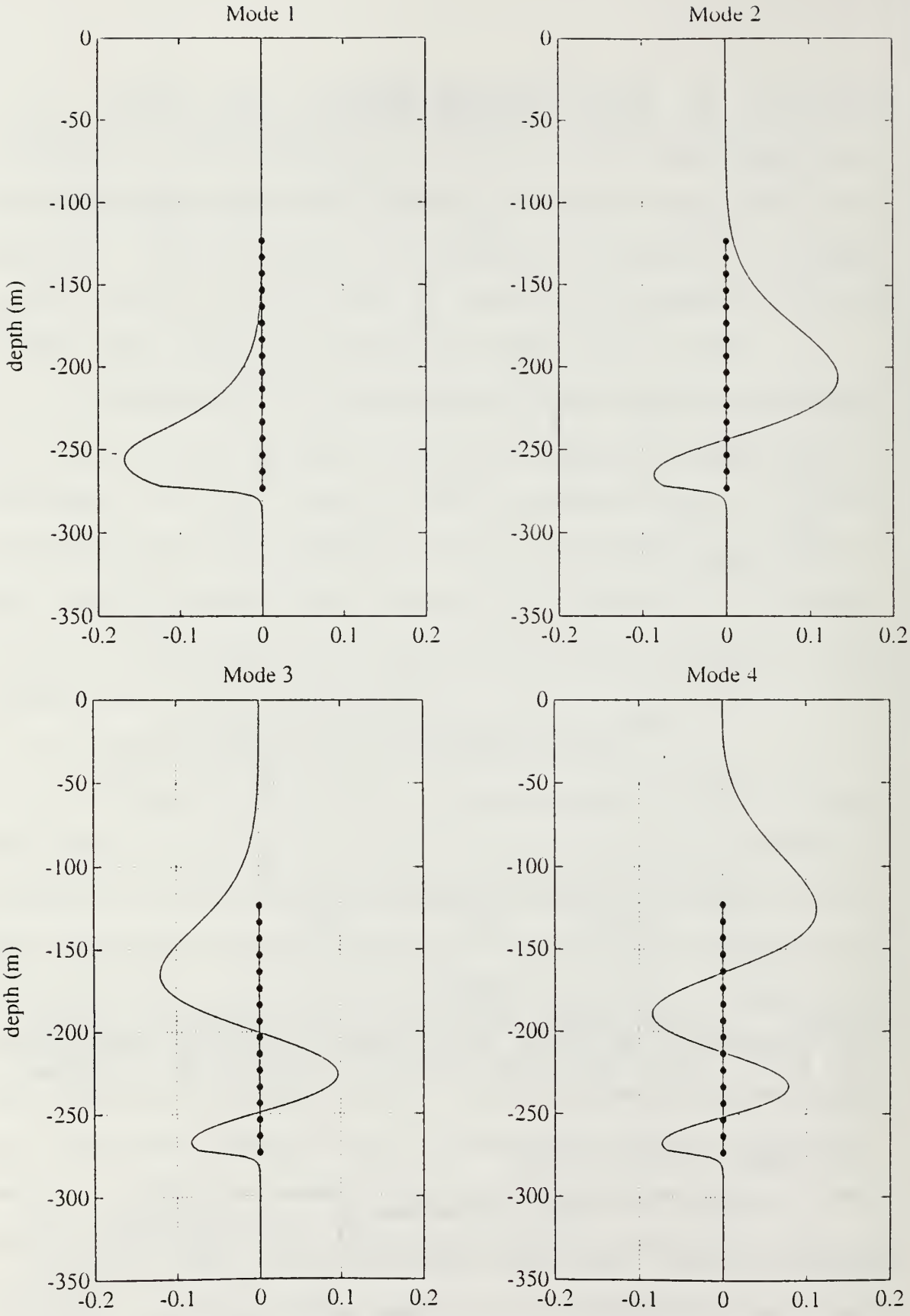

Figure 8 Normal modes 1 through 4.

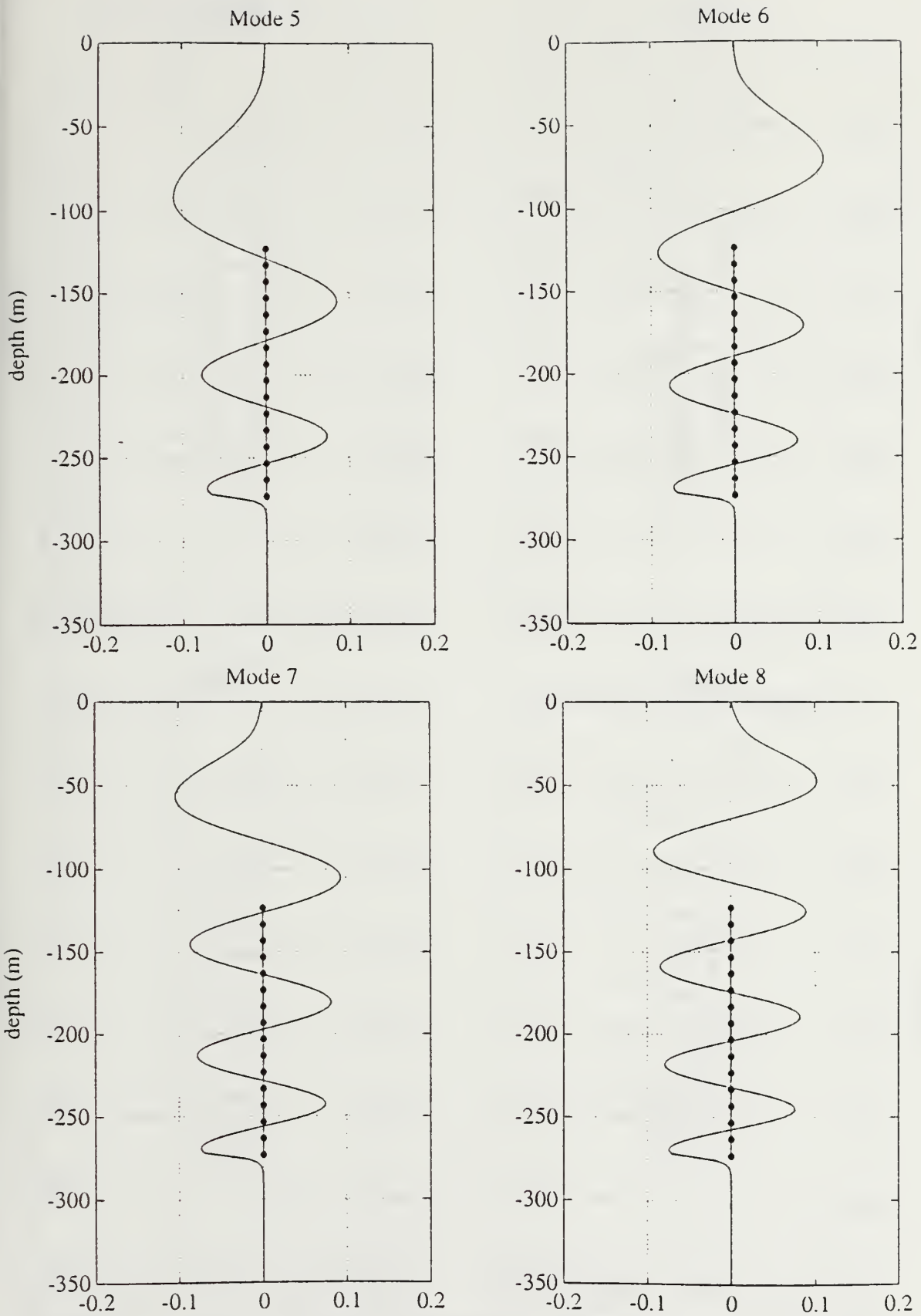

Figure 9 Normal modes 5 through 8.

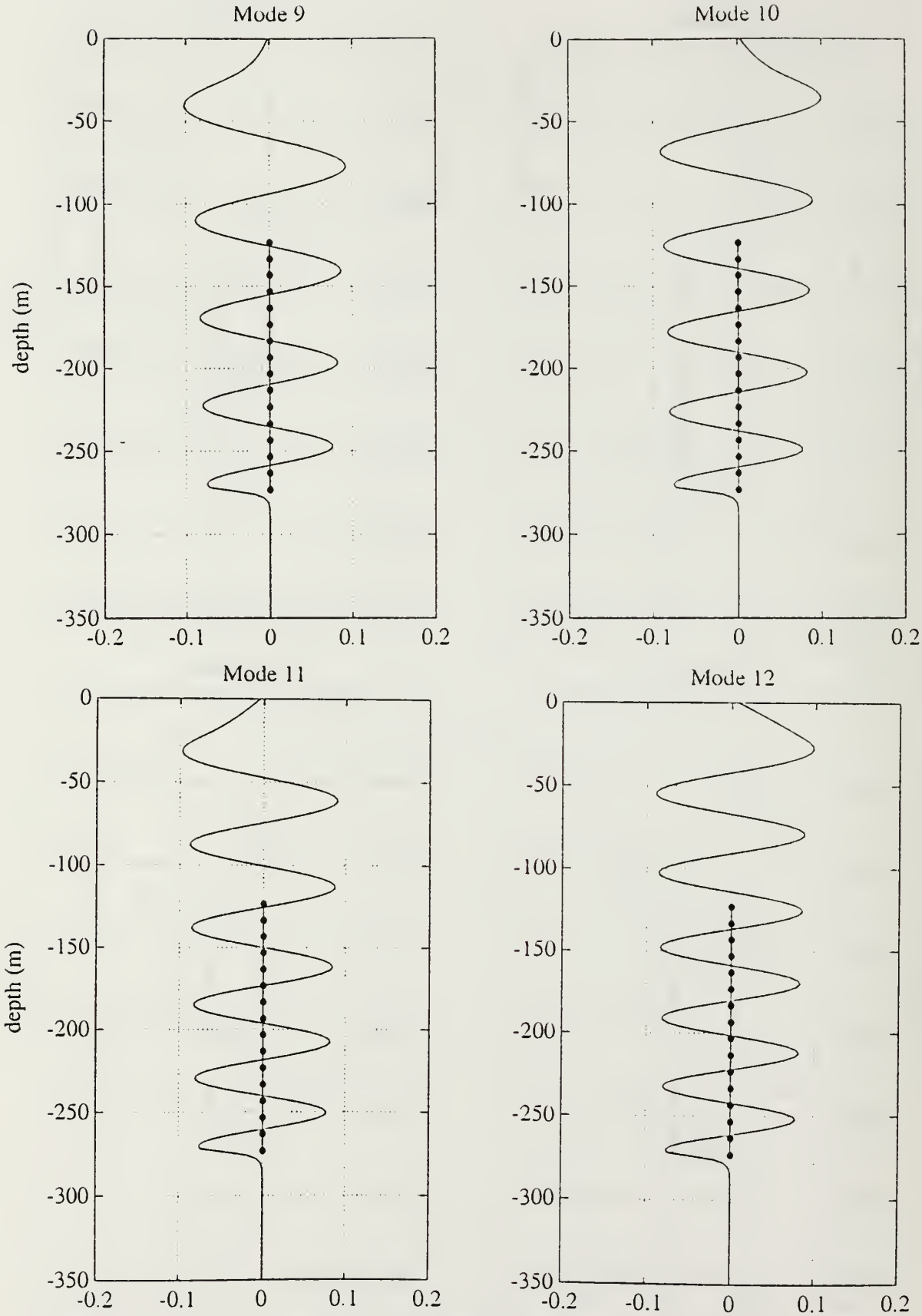

Figure 10 Normal modes 9 through 12.

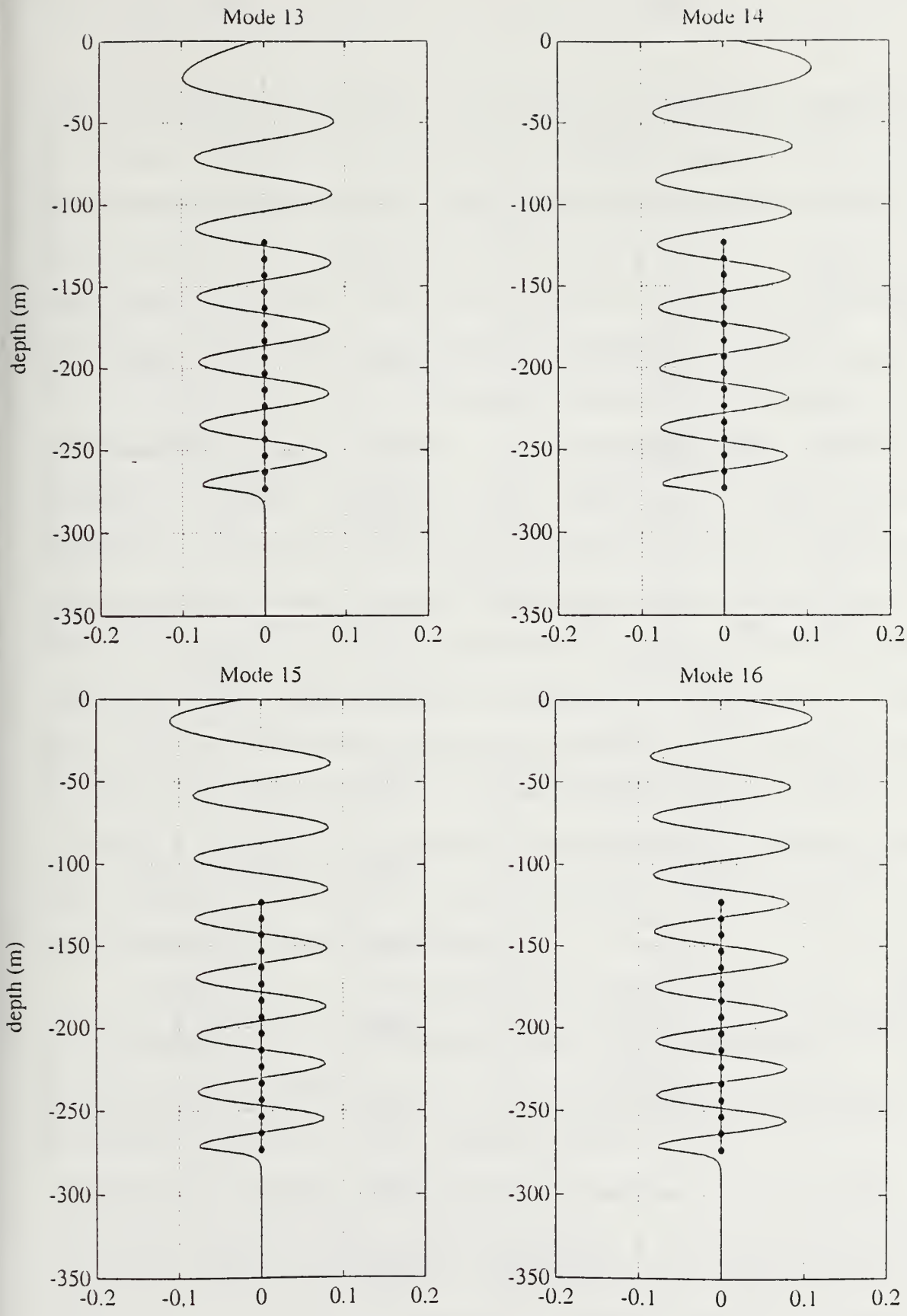

Figure 11 Normal modes 13 through 16.

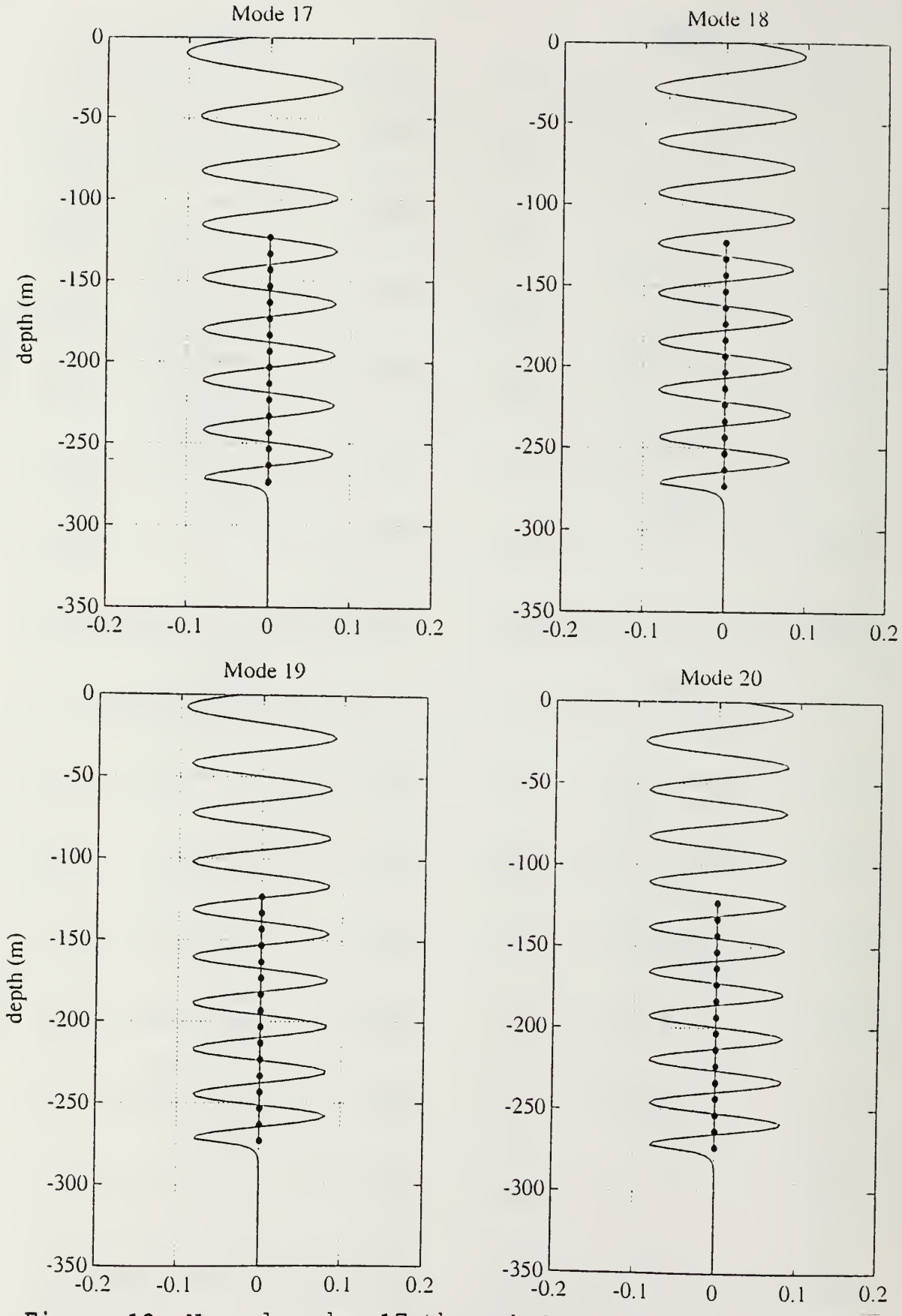

Figure 12 Normal modes 17 through 20.

easily determined through a multitude of available measures including: theoretical array gain, estimated array gain, and the steering beam pattern. Array gain experienced by modal beamforming cannot exceed the theoretical gain for a classical beamformer. However, there are additional issues to address concerning tomographic resolvability for mode arrivals.

One such factor, is the cross talk, or the recognition differential between modes. To quantify this affect, synthetic signals modeled from individual modes were beamformed. The modal signals are created using the frequency domain eigen solutions at each hydrophone depth. To duplicate the coded signals frequency structure, a *Blackman Window* is applied to the modal signal's frequency components. The square of the beamformed signal is proportional intensity of the mode. The peak values of the beamformed modes are squared and normalizing by the corresponding modal signal, forming the modal recognition kernel. The transpose of the recognition kernel gives the modal beam pattern.

Figures 13 to 18 are the modal beam patterns specific to the Barents Sea array geometry and ocean structure. The following is a summary of my conclusions and conjectures on the Barents Sea array modal beam pattern:

- Modes one through three are well resolved.
- Modes four and above are not spatially well resolved due to the size of the array's aperture.

- Above mode three resolvability must be handled on a case to case bases, taking into account the arrival structure and relative intensity of each mode.

- Assuming there is no appreciable modal energy above mode twenty, spatial aliasing effects are non-existent for this array.

- Above mode four, there is a recurring beam pattern which indicates that the closest modes are the hardest for the beamformer to differentiate. Mode four beam pattern shows the transition from well resolved to less resolvable arrivals.

- Increasing array length to 50 meters below the surface would increase the coverage of the array's aperture and guarantee that at least the first six modes are spatially resolved. This can be accomplished through increasing the number of hydrophones, or by expanding element spacing, subject to spatial aliasing.

Figure 13 Recognition beam pattern for modes one through four.

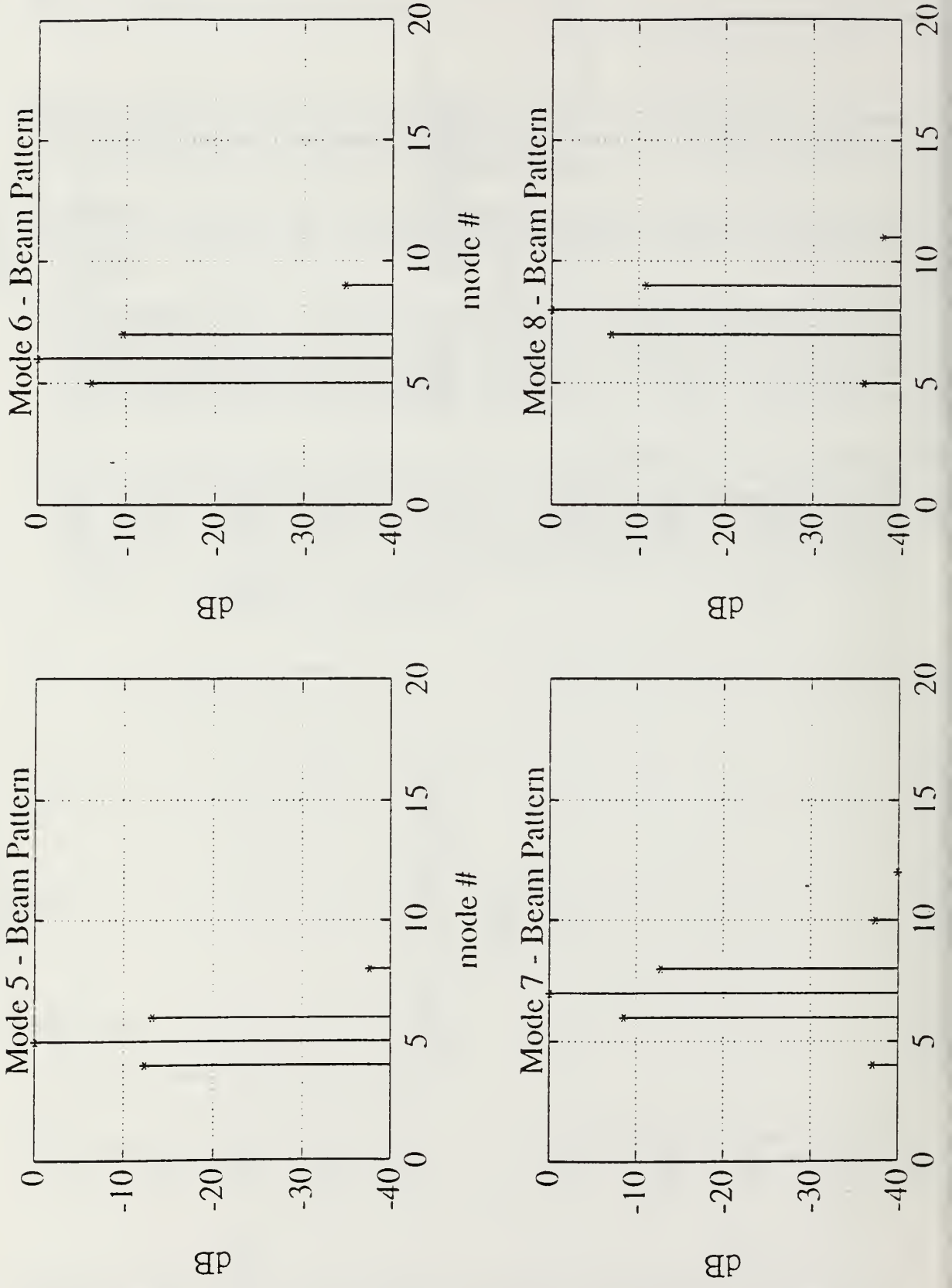

Figure 14 Recognition beam pattern for modes five through eight.

Figure 15 Recognition beam pattern for modes nine through twelve.

Figure 16 Recognition beam pattern for modes thirteen through sixteen.

Figure 17 Recognition beam pattern for modes seventeen through twenty.

VI. EVALUATION OF BARENTS SEA DATA

A. DECODING OF RAW ACOUSTIC DATA

The participants of the Barents Sea Polar Front Experiment performed some near-real-time signal processing during the experiment. Some of the signal processing included decoding of the 224 Hz source. Figure 18 is the decoded signal taken from the bottom hydrophone of the array.

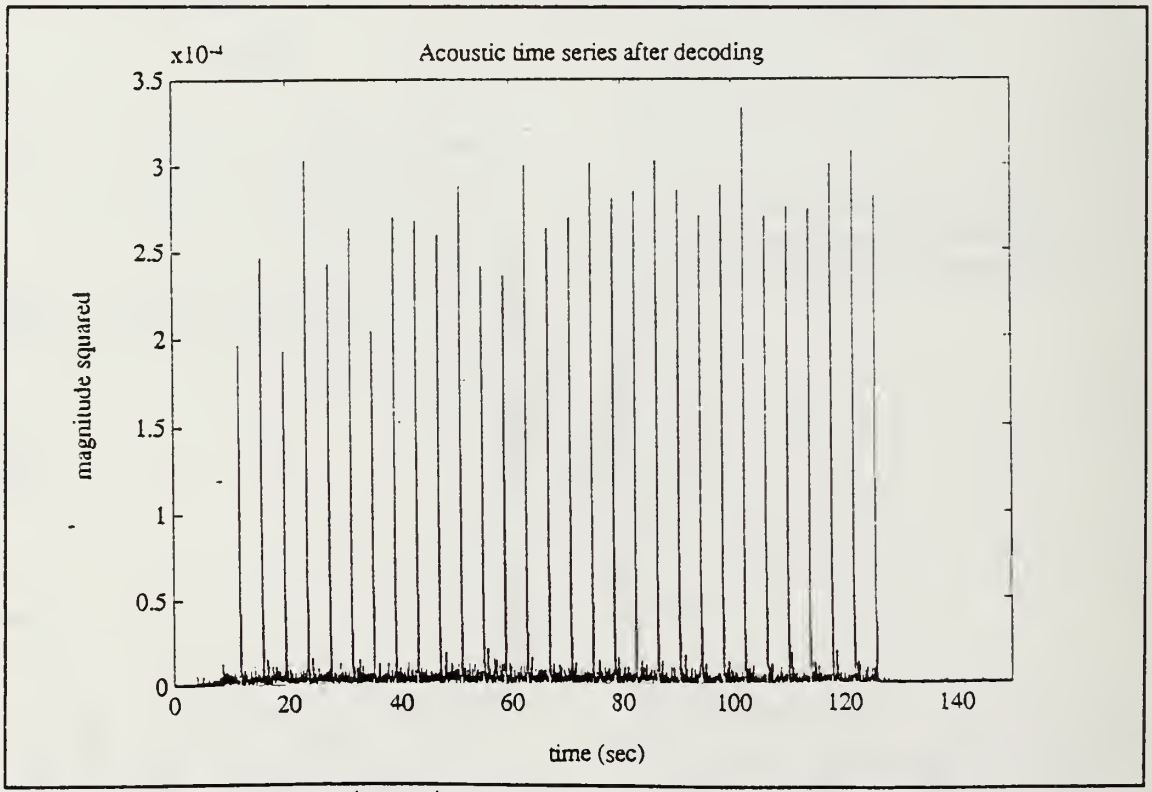

Figure 18 Acoustic time series after BM decoding for the bottom hydrophone in the array [Ref. 7].

The BM sequence removal algorithm performed very well. Theoretical decoding gain is based on the number of digits in

the coded signal. The 224 Hz source signal has a theoretical decoding compression gain of 18 dB which compares favorably to initial estimates. Signal gain estimates where determined by,

$$GAIN = 10\log\left(\frac{SNR_{after}}{SNR_{before}}\right), \qquad (6.1)$$

[Ref. 15].

Coherent averaging is a useful method for determining the acoustic arrival structure across the array. Figure 19 is the coherently averaged signal across the array, post m-sequence removal. The coherently averaged signal indicates the presence of at least two separate arrivals. This effect is most likely do to the time spreading of the signal between the propagation paths. The first arrival has the bulk of acoustic energy and a complex structure. Clearly this energy is formed by the bottom bounce propagation path. The second arrival lacks the energy and fine definition of the first caused by increased bottom and surface interaction.

B. RESULTS FROM THE BROADBAND MODAL BEAMFORMER

The results from broadband modal beamforming of the Barents Sea data are quite impressive. The mode arrival shapes are very distinctive, which simplifies arrival time estimation. All times for Figures 21-27 are relative to the record tape, and do not represent the travel from source emission. Figure 20 shows the mode one arrival structure for

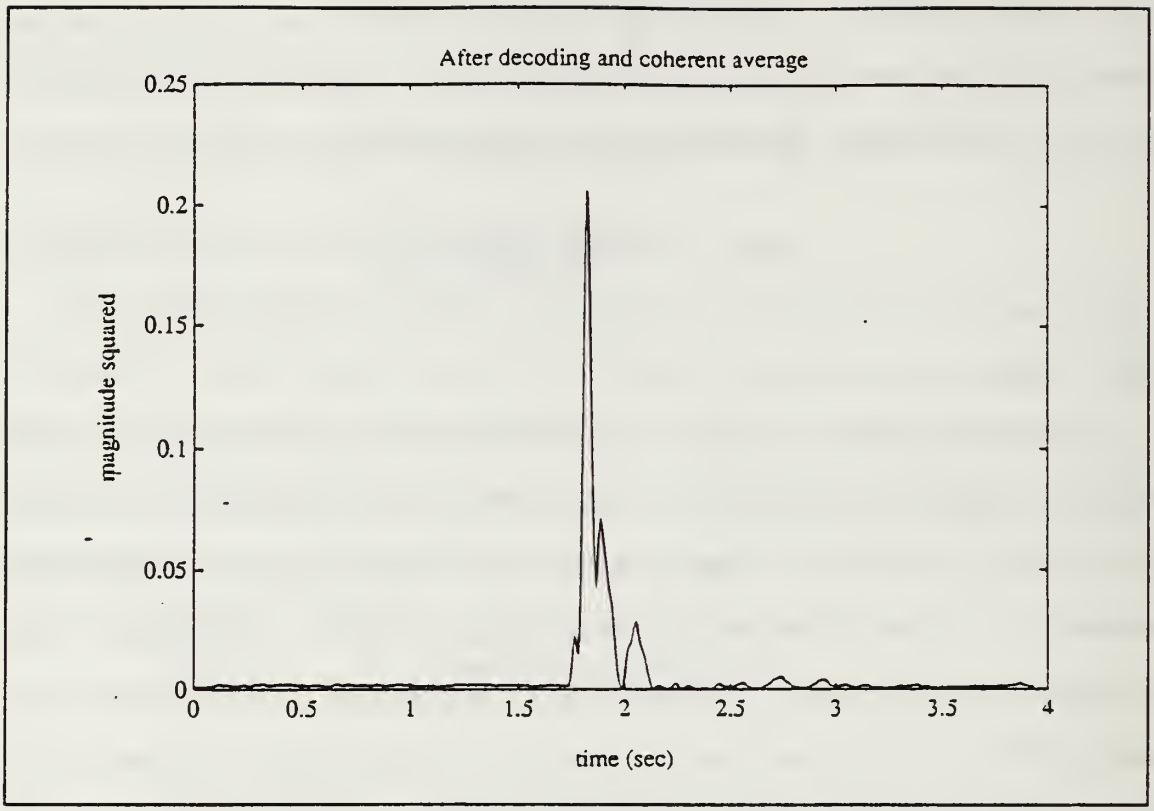

Figure 19 Coherent averaging after BM decoding for the WHOI-NPS array [Ref. 7].

the first four sequences of decoded pulses. Figures 21 through 23 are the first five modes pertaining to the first four sequence arrivals of Figure 18. The first five modes are the strongest arrivals. This is expected because the majority of modal energy is within the array aperture. Mode three is the strongest arrival indicating the majority of acoustic energy is in the lower 125 meters of the water column. Mode two is particularly interesting due to its multiple arrivals. This pattern might be attributed to coupling caused by the sharp change in sound speed gradient at the front. The mode two arrival is both the smallest in total energy and the earliest arrival of the first five beamformed modes.

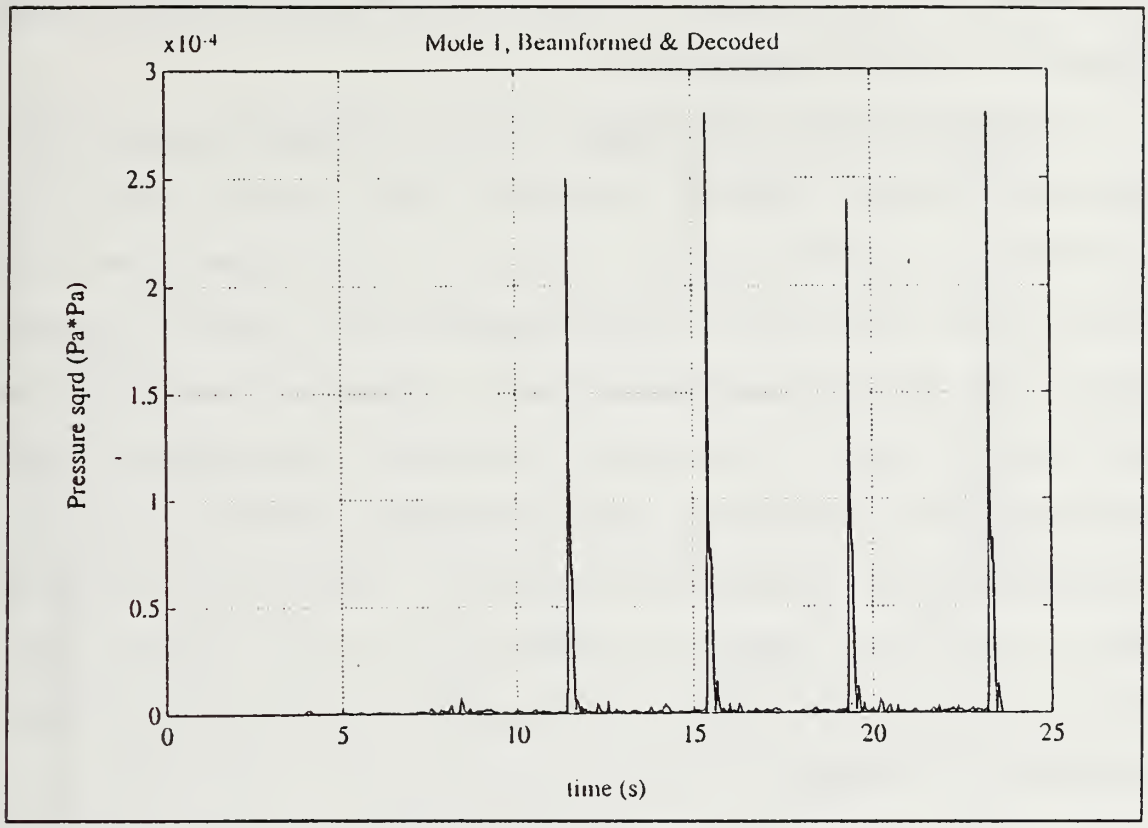

Figure 20 Mode one, beamformed and decoded arrival structure.

Modes six through ten, Figure 25, appear after the initial arrival structure. These modes contain less energy and have later arrival times. A large amount of modal energy exists outside the array aperture and is at least a partial factor to the decrease in beamformed energy.

Modes eleven through twenty, Figures 26 and 27, show a sharp drop off in energy levels which cannot be completely contributed to the array aperture. This sudden decrease of beamformed energy is most likely caused by high transmission loss caused by surface scattering. It is interesting to note that arrivals for modes sixteen and seventeen indicate

increased energy levels. This is a curious result that presently eludes explanation.

Broadband modal beamforming of the BSPFEX data provided extremely clear arrival structure that could easily be mistaken for modeled data. Part of the reason that the results are this good is the extremely high signal to noise ratios and the low propagation loss. Clearly, the first ten modes are the most significant to acoustic tomography. Mode resolvability is dependent upon the mode's arrival time, the modal beam pattern and spatial aliasing. Resolvability issues have not yet been completely sorted out but may indicate that only the first few modes are reliable enough to use in the tomographic inverse.

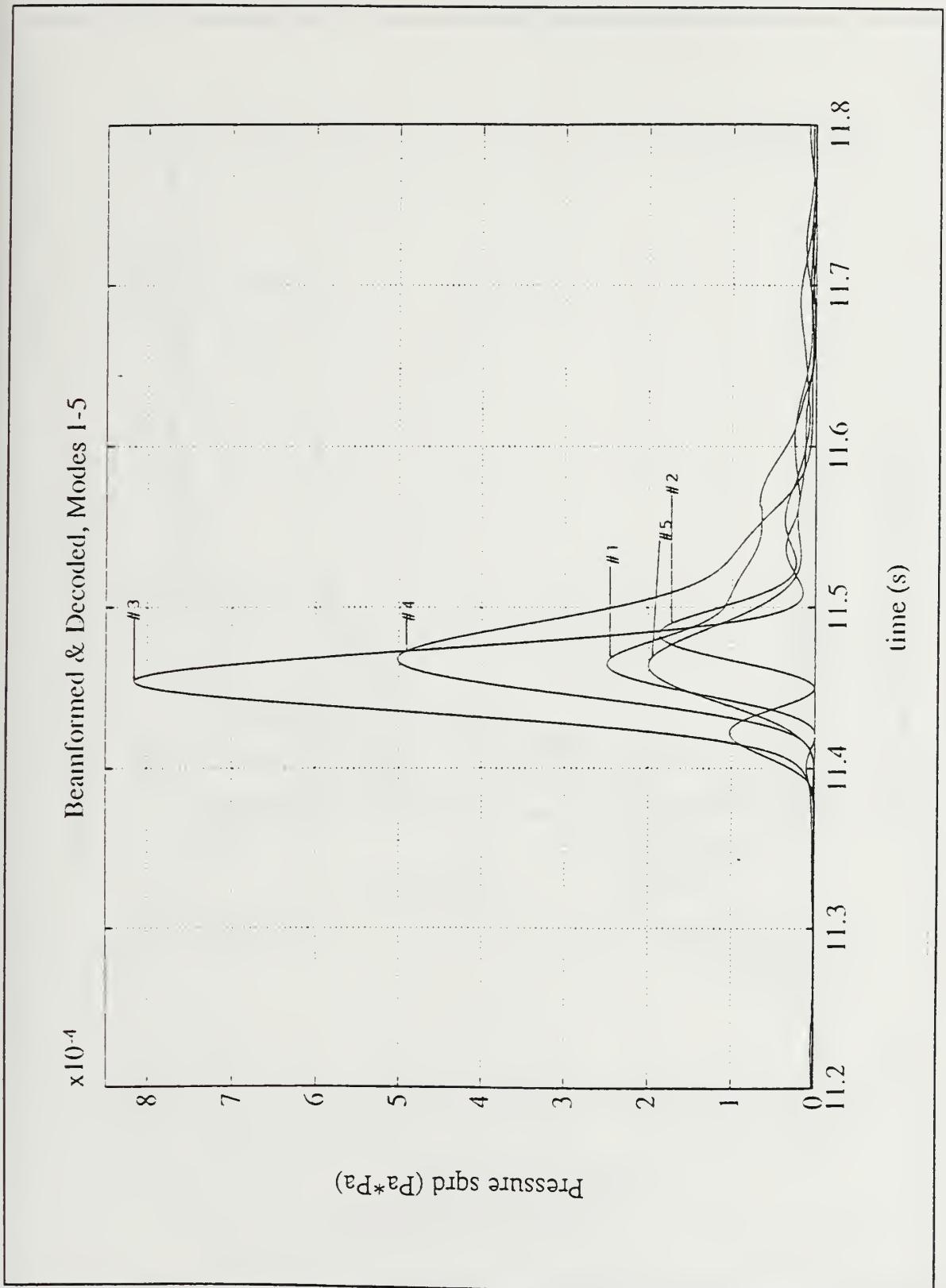

Figure 21 First arrival sequence for modes one through five.

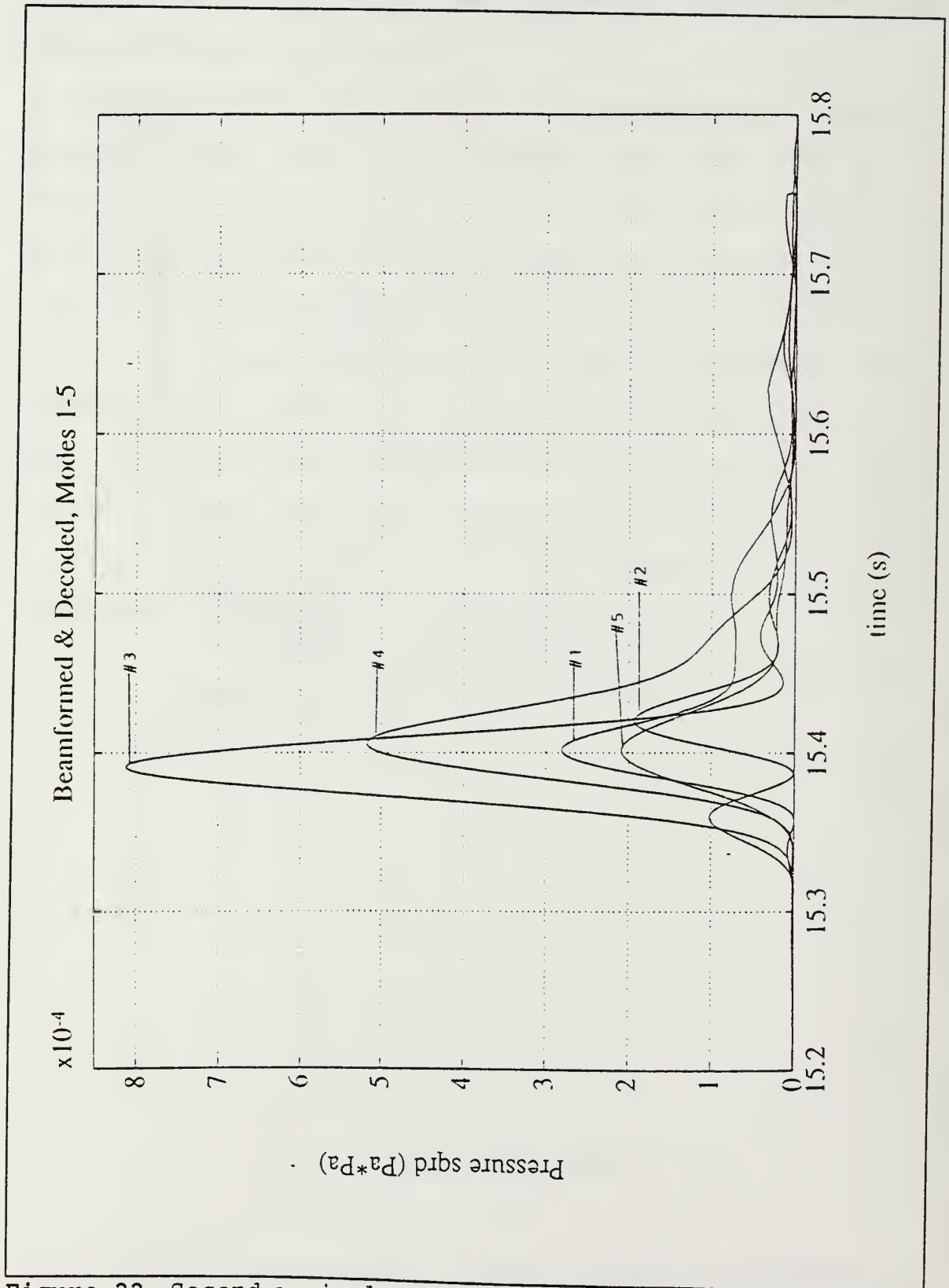

Figure 22 Second arrival sequence for modes one through five.

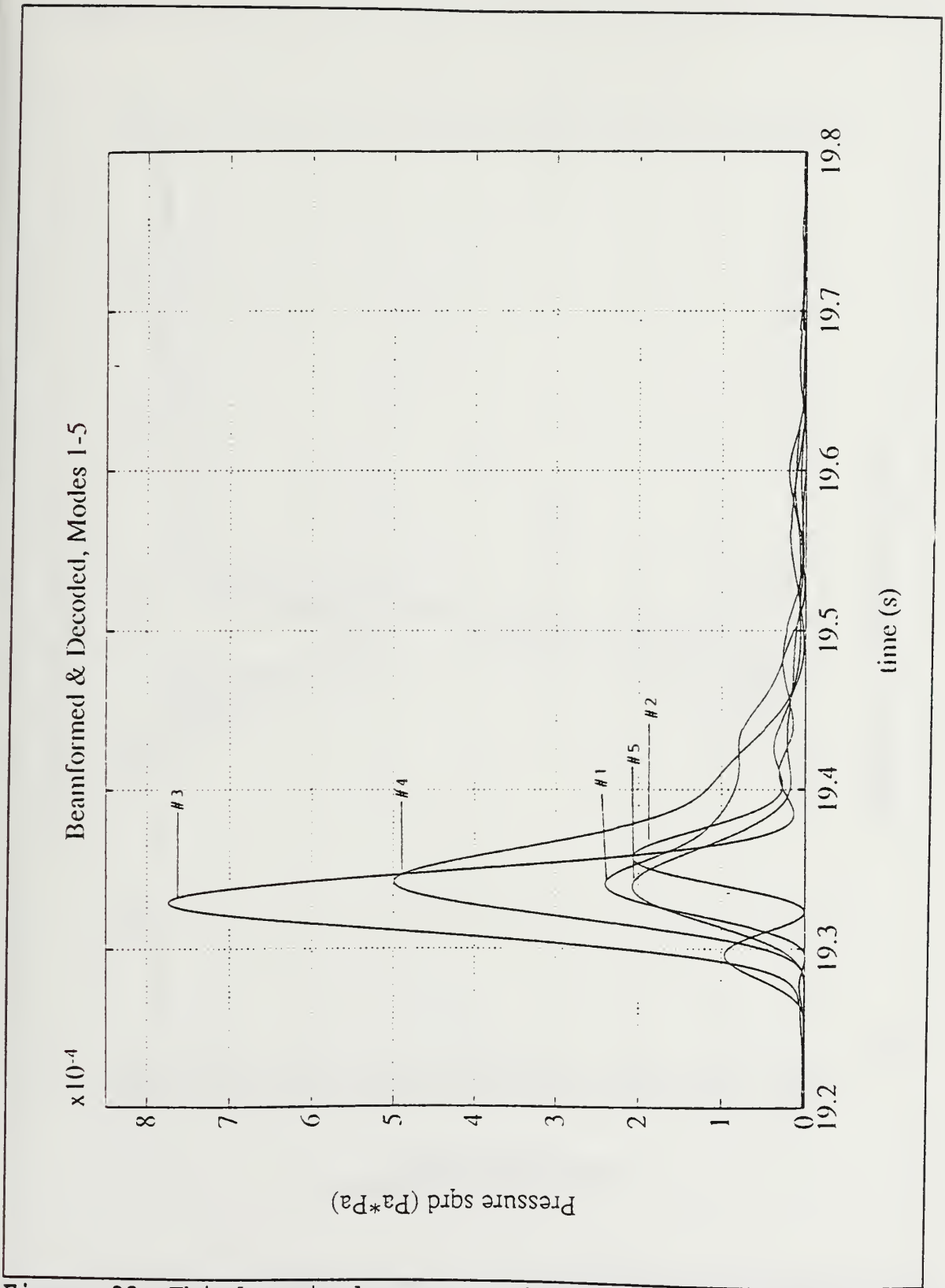

Figure 23 Third arrival sequence for modes one through five.

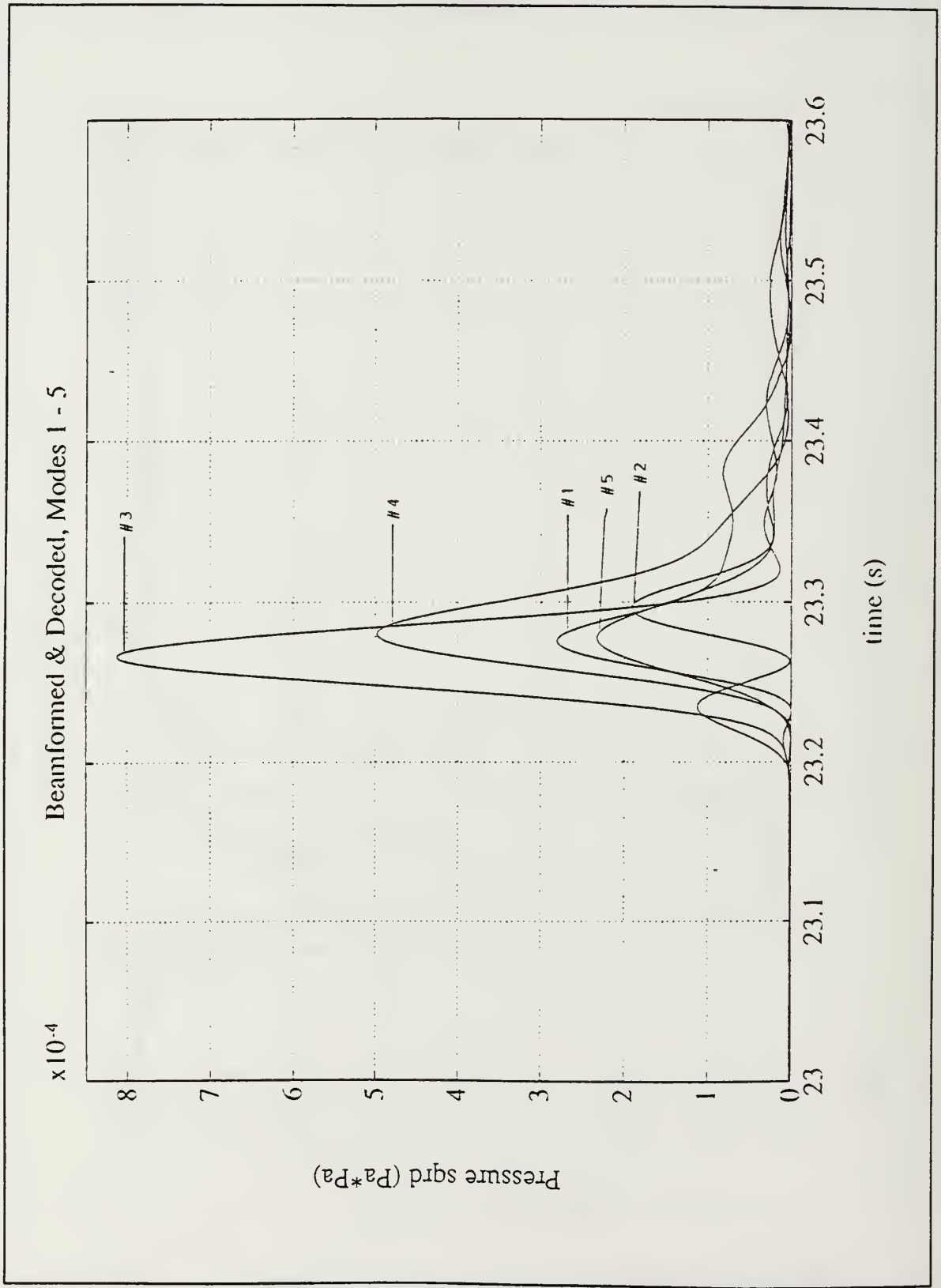

Figure 24 Fourth arrival sequence for modes one through five.

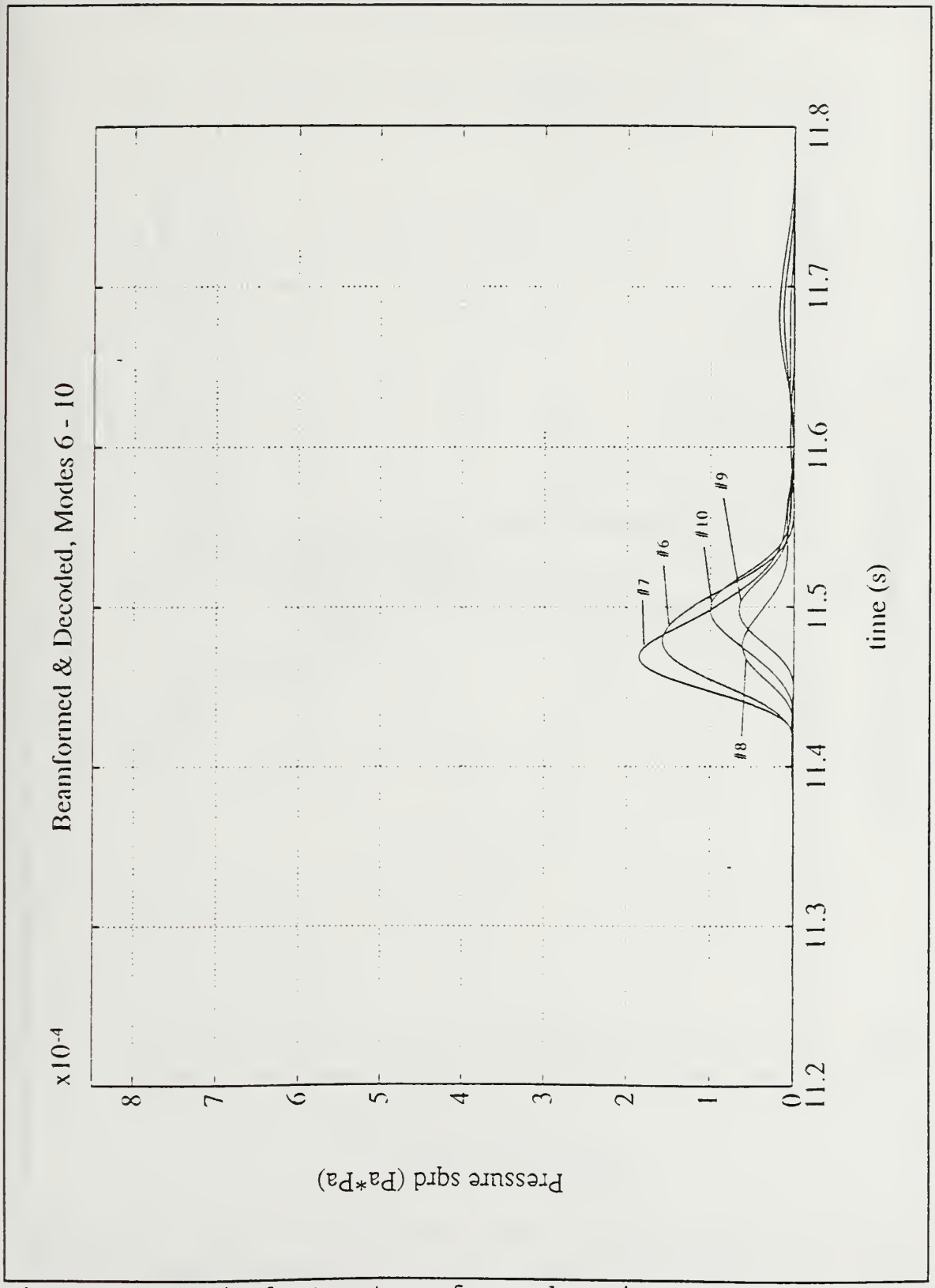

Figure 25 Arrival structure for modes six through ten.

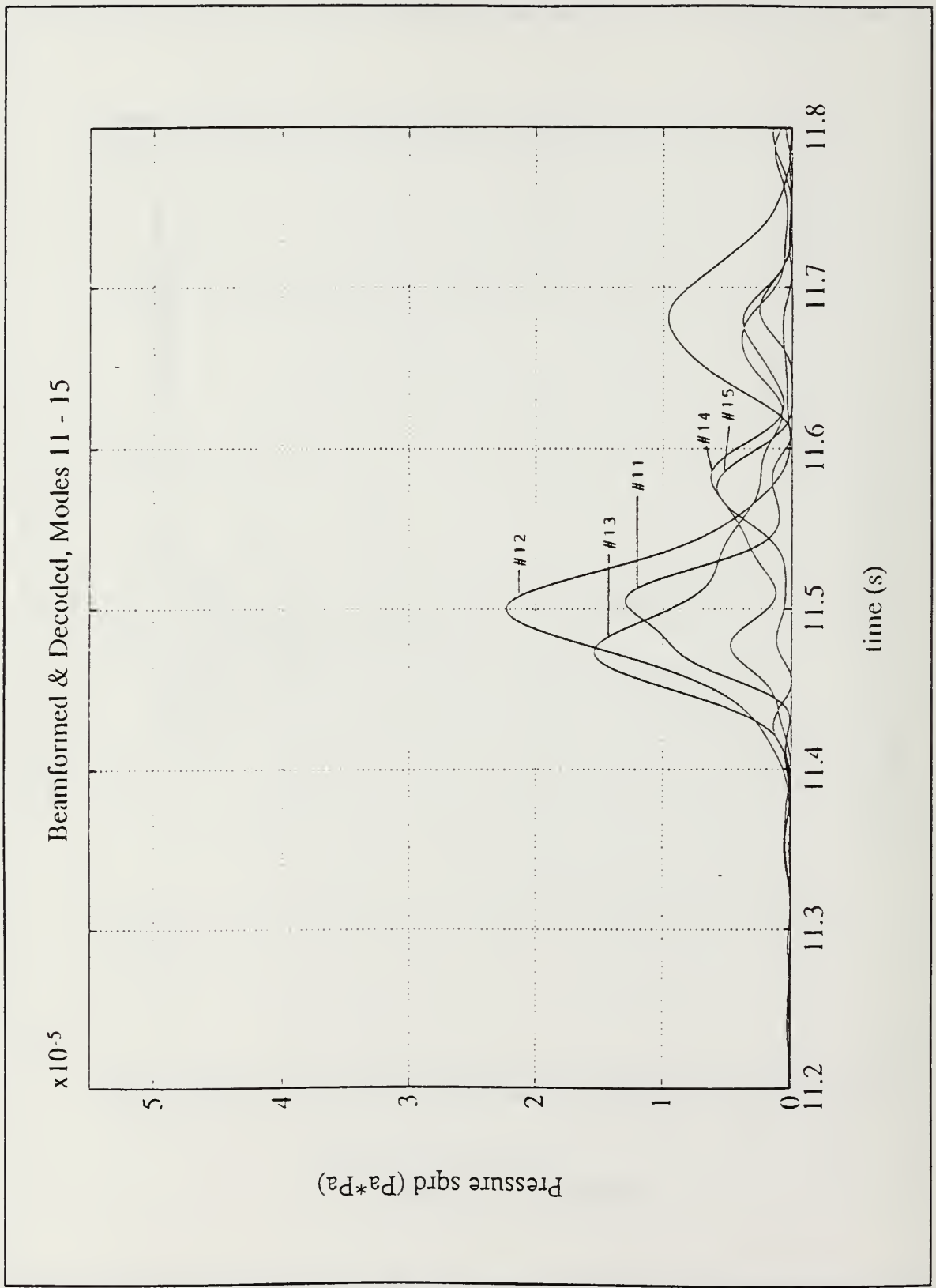

Figure 26 Arrival structure for modes eleven through fifteen.

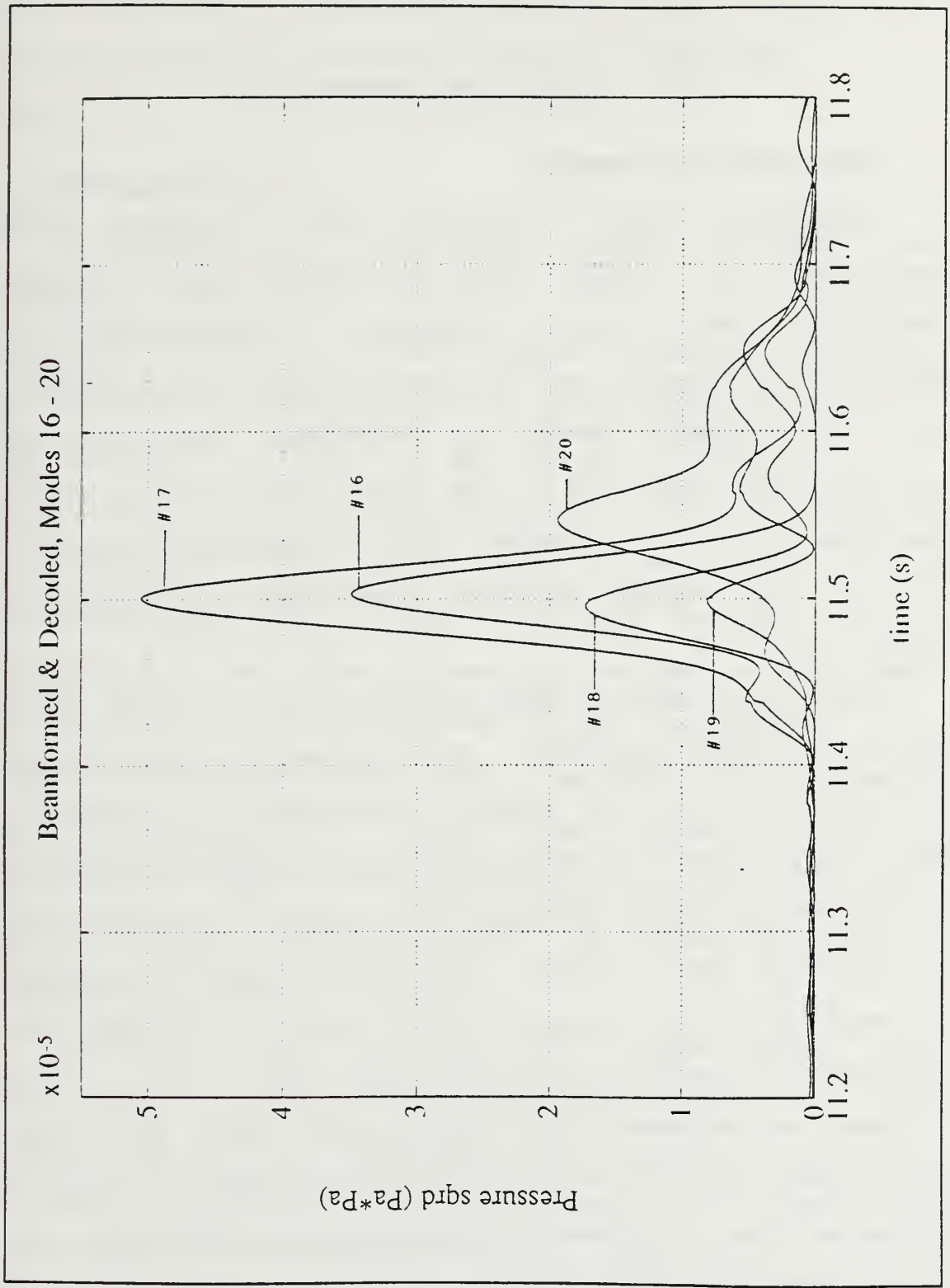

Figure 27 Arrival structure after beamforming for modes fifteen through twenty.

VII. SUMMARY AND FUTURE WORK

A. CONCLUSIONS AND SUMMARY

Preliminary results indicate that broadband modal beamforming is a useful tool in shallow water acoustic tomography. Ray acoustic tomography in shallow water is complicated because the earliest ray arrivals, containing the bulk of the acoustic energy, are unresolved. Unresolved ray paths are well described by the lower mode arrivals of the beamformed array. Broadband modal beamforming produces distinct mode energy arrivals simplifying estimation of arrival times.

Improvements made by WHOI and NPS to the vertical array design proved very successful. The Dual Benthos interrogators provided a more reliable source for measuring array geometry than previous methods. Array stability was vastly improved by its new moored design and should be used as a benchmark for the design of future arrays. The measured array tilt during the experiment was a nominal three tenths of a degree and minimized the correction required for array tilt.

Based on recommendations from Crocker [Ref. 4], a fast third order interpolation routine was implemented to reduce processing time. The new interpolator was derived using the *Taylor Approximation* for a third order polynomial [Ref. 16].

This algorithm is approximately five times faster than previous interpolator based on *Neville's algorithm* [Ref. 17].

B. RECOMMENDATIONS

The broadband modal beamformer requires each mode to be processed individually. Most of the time incurred from processing is due to the Fast Fourier Transform algorithm. Processing all modes simultaneously would reduce the number of Fast Fourier Transforms by the number of desired modes. Significant savings in processing can be realized by adopting this technique. Calculating the modal beam pattern was an additional time sink. Since each mode pattern must be processed for every mode, processing time goes as the square of the number of modes desired in the beam pattern. For twenty modes, the beamformer must be run four hundred times to get the beam pattern for just one sound velocity profile. If all the modes where processed simultaneously, the modal beam pattern could be an added option to the program reducing the processing required by the user.

Decoding of the m-sequence coded signal occurs after beamforming. The decoding process could have been accomplished prior to beamforming but would have increased processing time by the number of elements in the array. The output from the m-sequence removal algorithm is complex numbers thereby doubling the amount of memory required to

store data. Despite this, it may still be desirable to decode the data first so that plane wave and modal beamforming can be accomplished using the same data base. To perform sequence removal prior to beamforming, the decoder algorithm must be adapted to handle multi-channel data, and an interpolation routine should be added to resample data prior m-sequence removal. The modal beamformer will also need to be updated to handle complex input and to perform mode filtering on demodulated signals.

Design of the next vertical array should incorporate estimations of the modal beam pattern. Modeling of the acoustic environment will give an estimate of the modal beam pattern. Using this information, the number of array elements and geometry can be optimized to achieve the desired performance level.

APPENDIX

A. BROADBAND MODAL BEAMFORMER ALGORITHM

The broadband modal beamformer was designed for a robust environment. Program portability allows it to be used on any computer system supporting C or quick C software. The program is flexible enough to cope with most vertical line array designs. A large number of user parameters are incorporated in the computer architecture to enhance portability. The following is a summery of user parameters employed.

1. Program Parameters

- UNIX VERSION: a compiler flag used to determine the operating system, either UNIX or ANSI.
- ASCII: ASCII and BINARY options determine the beamformed output type. These options are inter-linked, and never can be activated at the same time.
- SIGNAL: enables time series beamformed output. This parameter should be left "ON".
- LOWER_SENSOR: enables lower sensor tilt data if available.
- VALIDATE: this option flags the program to output several validation files. If selected, the program will output files containing array geometry, steering delays, hydrophone weights at the lowest frequency in the band, and estimated mode speed.
- ERROR_ESTIMATE: flags to produce the estimated error caused by interpolation. The error estimate is stored in a file specified by the user.
- VERBOSE: if selected, this option will cause the program to provide a status report during operation. Information is sent to standard output. This option is useful for debugging and for short data sets.
- ON/OFF: logical switches for program control.

- INTERPOLATE: specifies which interpolation routine to use. The available interpolators are fastpoly3, polint and ratint. Fastpoly3 is the fastest of the three interpolators. Other interpolators can be added by the user if desired.

- ORDER: determines the order for interpolation. Polint and ratint are multi order algorithms, while fastpoly3 can only perform third order interpolation. If the order of interpolation is not three and the fastpoly3 interpolator is selected, the program will terminate on a standard error.

- STEP: indicates the number of points on either side of a sequence for derivative estimates.

- TINY: smallest number allowed for floating point operations.

- PI: trigonometric constant.

- RADIAN: degree to radian conversion constant.

- OFFSET: correction term for the difference between tilt sensor depth and the depth of the first hydrophone.

- DELTA_R: array element spacing in meters.

- CTD_OFFSET: used to correct for offset caused by the CTD measurement. This offset is reflected in the eigen solutions.

- SAMPLE_DELAY: corrects for sampling skew caused by sampling data from a single data bus (ms).

- SSP_LENGTH: dimensioning term based on the maximum number of points in a single eigen function.

- EIGVAL_LENGTH: dimensioning term based on the maximum number of frequencies contain in the eigen value file.

- LOOK DIRECTION: desired direction for beamforming in degrees true.

- TILT_BUFFER: dimensioning factor based on the maximum number of tilt observations in the tilt input file.

- BUFFER_TIME: dimensioning term. Must be an integer and greater then the FFT_TIME + 2.

- F_SAMPLE: the integer sampling frequency.

- CHANNELS: total number of array elements.
- F_CARRIER: the desired carrier frequency to beamform. Note that this should match the center frequency of the eigen function and value input files.
- FFT_TIME: amount of data, in seconds, allotted to each FFT. This should be the sequence length of the desired beamformed signal.
- FFT_LENGTH: the radix two FFT size. The FFT_LENGTH must be larger or equal to FFT_TIME*F_SAMPLE.
- SWAP: macro used by the FFT algorithm.

2. Beamformer Source Code

```
/****
*  PROGRAM: BEAMFORMER vsn 5.0
*  NAME:    BBbeam.c
*  WRITTEN BY: Glenn A. Omans II / Steven Crocker
*
*  PURPOSE: Broadband Modal Beamforming of a Vertical Array.
*
*  LAST UPDATE: September 28, 1992
*
*
*  This program takes input from various data files and the user.  It
*  outputs a data file.  The inputs are a number of channels of digital
*  acoustic  data, and information regarding the physical characteristics
*  and geometry of the receiving array.  Additionally, environmental data
*  in the form of normal mode eigenfunctions and eigenvalues at the
*  receiving array are required to operate this beamformer.  The output
*  is a single channel of acoustic data.
*
****/
#define UNIX VERSION              /* either ANSI or UNIX                  */
#define ASCII ON                  /* select output mode ON or OFF         */
#define BINARY OFF                /* select output mode ON or OFF         */
#define SIGNAL ON                 /* either ON or OFF                     */
#define LOWER_SENSOR OFF          /* either ON or OFF                     */
#define VALIDATE OFF              /* either ON or OFF                     */
#define ERROR_ESTIMATE OFF        /* either ON or OFF                     */
#define VERBOSE OFF               /* either ON or OFF                     */
#define ON 1                      /* logical "switch"                     */
#define OFF 0                     /* logical "switch"                     */
#define INTERPOLATE fastpoly3     /* either polint, fastpoly3 or ratint   */
#define ORDER 3                   /* order of interpolator          (odd)*/
#define STEP 1                    /* number of steps for derivatives      */
#define TINY 1.0e-25              /* prevents division by zero            */
#define PI 3.14159265359          /* for freq to omega conversions        */
#define RADIAN 57.2957795131      /* for degree to radian conversions     */
#define OFFSET 0.0                /* dist btwn upper sensor and phone #1  */
#define DELTA_R 10.0              /* array element spacing                */
```

```
#define CTD_OFFSET 0.0          /* diff btwn ctd depth inc & 1st depth
#define SAMPLE_DELAY 0.003      /* array sampling delay (ms)
#define SSP_LENGTH 2500         /* max number of pts in eigunfunction
#define EIGVAL_LENGTH 230       /* max number of eigenvalues
#define LOOK_DIRECTION  270.0   /* direction from which signal arrives
#define TILT_BUFFER 120         /* max length of tilt data vectors
#define BUFFER_TIME 7           /* input buffer length in seconds (int)
#define F_SAMPLE 1600           /* sampling frequency             (int)
#define CHANNELS 16             /* number of channels processed
#define F_CARRIER 224.0         /* carrier frequency
#define FFT_TIME 3.9375         /* time alloted to each fft
#define FFT_LENGTH 8192         /* radix 2 <= (FFT_TIME-2)*F_SAMPLE
#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

#include<stdio.h>
#include<malloc.h>
#include<math.h>

#if defined ( ANSI )
#include<float.h>
#include<stdlib.h>
int getInput(void);
int putOutput(void);
int processTilt(float **x, float **y, float **z, float *delayClock);
int processModes(float **z, float **weight, float *ptrC);
int dydx(float *x, float *y, float *ddx, int points);
int fastpoly2(float *xa, float *ya, int n, float x, float *y, float *dy);
int fastpoly3(float *xa, float *ya, int n, float x, float *y, float *dy);
int polint(float *xa, float *ya, int n, float x, float *y, float *dy);
int ratint(float *xa, float *ya, int n, float x, float *y, float *dy);
int realft(float *data, int n, int isign);
int four1(float *data, int nn, int isign);
int window(float *data, int N);
float *vector(int length);
float **matrix(int row, int col);
int free_matrix(float **m, int row);
void ExitOnError(char error_txt[]);
#elif defined ( UNIX )
int   vector(),
      matrix(),
      getInput(),
      putOutput(),
      processTilt(),
      processModes(),
      dydx(),
      fastpoly2(),
      fastpoly3(),
      polint(),
      ratint(),
      dumpSpectrum(),
      window(),
      realft(),
      four1(),
      free_matrix(),
      ExitOnError();
#endif

   /* Global Variable Declarations */
float **inSOUND, *outSOUND, *MeanSqError, Max=0.0, Min=1.0e25;
float fmax, fmin;
int LastDelay, Minute=1, firstBuffer=1;
```

```
FILE *fpInSound, *fpOutSound, *fpOutSpectrum, *fpMSE;

main()
{
   char fileDelay[80], fileModes[80], fileArray[80];

   FILE *fpV1;

   int i, j, k, n, itime, ibuff, kount, *shift;

   float **x, **y, **z, *indx, *samples, arg, ans, err, *delay,
         *delta, **weight, *pwrSpectrum, Cgroup, *Xf,
               *sumXf, **buffSOUND, df, freq, *delayClock;

    double Time;


   /* Memory Allocation and Tilt Data Processing */
   x=(float**)matrix(CHANNELS, TILT_BUFFER);
   y=(float**)matrix(CHANNELS, TILT_BUFFER);
   z=(float**)matrix(CHANNELS, TILT_BUFFER);
   delayClock=(float*)vector(TILT_BUFFER);

   processTilt(x, y, z, delayClock);

/* printf("\nDelay Clock Times\n\n");
   for (i=0; i<=LastDelay; i++) printf("%i\t%g\n",i,delayClock[i]);  */

   if (VALIDATE==ON)
   {
      printf("\nEnter file name for array validation data: ");
      if((scanf("%s",fileArray))==EOF)
         ExitOnError("Fatal error in scanf()");

      fpV1=fopen(fileArray,"wt");
      if(fpV1==NULL) ExitOnError("Error opening validation file");

      fprintf(fpV1,"Channel\t\t X\t\t Y\t\t Z\n");

      for (i=1;i<=LastDelay;i++)
      {
         fprintf(fpV1, "MINUTE: %i\n", i);
         for (j=1;j<=CHANNELS;j++)
                  fprintf(fpV1,    "%i\t    %f\t    %f\t
%f\n",j,x[j][i],y[j][i],z[j][i]);
      } /* for */
      fclose(fpV1);
    } /* if */

   /* Memory Allocation*/
   weight=(float**)matrix(CHANNELS,FFT_LENGTH);
   indx=(float*)vector(ORDER+1);
   delay=(float*)vector(CHANNELS);
   delta=(float*)vector(CHANNELS);
   outSOUND=(float*)vector((BUFFER_TIME-1)*F_SAMPLE);
   Xf=(float*)vector((FFT_LENGTH+1)*2);
   sumXf=(float*)vector((FFT_LENGTH+1)*2);
   buffSOUND=(float**)matrix(CHANNELS,(BUFFER_TIME-1)*F_SAMPLE+1);
   inSOUND=(float**)matrix(CHANNELS, BUFFER_TIME*F_SAMPLE);
   if ((shift=(int*)malloc((CHANNELS+1)*sizeof(int)))==NULL)
      ExitOnError("Memory allocation failure for shift[].");
```

```
   for (i=1; i<=ORDER+1; i++) indx[i]=(float)i;
        i f      ( E R R O R _ E S T I M A T E = = O N )
MeanSqError=(float*)vector((BUFFER_TIME-1)*F_SAMPLE);

   /* Mode Data Processing */
   processModes(z, weight, &Cgroup);

   /* Calculate delays, shifts, ect */
   for (i=1; i<=CHANNELS; i++)
   {
      delay[i]=x[i][Minute]/Cgroup +                /* Time delay   */
               (float)(i-1)*SAMPLE_DELAY/1000.;     /* Sampling delay */
      shift[i]=(int)(delay[i]*(float)F_SAMPLE);     /* # of samples */
                                                    /* fraction of 1 sample
*/
      delta[i]=delay[i]*(float)F_SAMPLE-(float)shift[i];
   } /* for */

   while(Minute<=LastDelay)
   {

      if (VALIDATE==ON)
      {
         if(firstBuffer)
         {
            printf("\nEnter file name for delay validation: ");
            if((scanf("%s",fileDelay))==EOF)
               ExitOnError("Fatal error in scanf()");

            if ((fpV1=fopen(fileDelay, "wt")) == NULL)
               ExitOnError("Error opening validation file.");
          } /* if */
         else
         {
            if ((fpV1=fopen(fileDelay, "at")) == NULL)
               ExitOnError("Error opening validation file.");
         } /* else */

         fprintf(fpV1,"MINUTE: %i\n", Minute);

         fprintf(fpV1,
                "Channel\t delay\t\t int shift\t fraction of 1 shift\n");

         for (i=1;i<=CHANNELS;i++)
         {
            fprintf(fpV1, "%i\t %e\t %i\t% e\n",
                  i,delay[i],shift[i],delta[i]);
         } /* for */
         fclose(fpV1);

         if(firstBuffer)
         {
            printf("\nEnter file name for phone weights and group speed:
");
            if((scanf("%s",fileModes))==EOF)
               ExitOnError("Fatal error in scanf()");

            if ((fpV1=fopen(fileModes, "wt")) == NULL)
               ExitOnError("Error opening validation file.\n");
             fprintf(fpV1,    "Group    speed    for    F_CARRIER    is:
%g\n\n\n",Cgroup);
```

```
         } /* if firstBuffer */
         else
         {
            if ((fpV1=fopen(fileModes,"at")) == NULL)
               ExitOnError("Error opening validation file.\n");
            fprintf(fpV1,"\nHydrophone weights for minute %i\n",Minute);
         } /* else if not firstBuffer */

         fprintf(fpV1, "Channel \tWeight\n");

         for (i=1;i<=CHANNELS;i++)
            fprintf(fpV1, "%i\t%e\n",i,weight[i][1]);

         fclose(fpV1);
      } /* if VALIDATE */

      Time = 1;                                  /* First second of time is
chopped */
      for (n=1; n<=60/(BUFFER_TIME-2); n++)   /* for one Minute */
      {
         getInput();

         if(firstBuffer)    /* Produce first output buffer */
         {
            if (VERBOSE) printf("Interpolation first Buffer\n");
            for (i=1; i<=F_SAMPLE; i++)
            {
               outSOUND[i]=0.0;
               if (ERROR_ESTIMATE==ON) MeanSqError[i]=0.0;
            } /* for i */
            for (i=F_SAMPLE+1; i<=(FFT_TIME+1)*F_SAMPLE; i++)
            {
               outSOUND[i]=0.0;
               if (ERROR_ESTIMATE==ON) MeanSqError[i]=0.0;

               Time+=(1/(float)F_SAMPLE);

               if (Time >= delayClock[Minute])
               {
                  Minute++;

                  /* Mode Data Processing */
                  processModes(z, weight, &Cgroup);

                  /* Calculate delays, shifts, ect */
                  for (k=1; k<=CHANNELS; k++)
                  {
                     delay[k]=x[k][Minute]/Cgroup +              /* Time
delay   */
                              (float)(k-1)*SAMPLE_DELAY/1000.;       /*
Sampling delay */
                     shift[k]=(int)(delay[k]*(float)F_SAMPLE);   /* # of
samples */
                                                                    /*
fraction of 1 sample */
                     delta[k]=delay[k]*(float)F_SAMPLE-(float)shift[k];
                  } /* for */
               } /* if (Need new shift delays)  */

               for (j=1; j<=CHANNELS; j++)
               {
```

```
                if (delay[j]>=0.0)
                {
                   arg=(float)(ORDER+1)/2.0+(1-delta[j]);
                   samples = &inSOUND[j][i-shift[j]-(ORDER+1)/2];
                } /* if */
                else if (delay[j]<0.0)
                {
                   arg=(float)(ORDER+1)/2.0+delta[j];
                   samples = &inSOUND[j][i-shift[j]-(ORDER+1)/2+1];
                } /* else if */

                INTERPOLATE(indx, samples, ORDER+1, arg, &ans, &err);
                buffSOUND[j][i-F_SAMPLE]=ans;

                if (ERROR_ESTIMATE==ON)
                   MeanSqError[i]=MeanSqError[i]+err*err;
             } /* for */
             if (ERROR_ESTIMATE==ON)
                MeanSqError[i]=MeanSqError[i]/(float)CHANNELS;
          } /* for */
       } /* if */
       else      /* Produce subsequent output buffers */
       {
          if   (VERBOSE)   printf("Interpolating   subsequent   output
buffers\n");

          for (i=F_SAMPLE+1; i<=(FFT_TIME+1)*F_SAMPLE; i++)
          {
             outSOUND[i-F_SAMPLE]=0.0;
             if (ERROR_ESTIMATE==ON) MeanSqError[i-F_SAMPLE]=0.0;

             Time+=(1/(float)F_SAMPLE);  /* time processed with first
second chopped  */

             if (Time >= delayClock[Minute])
             {
                Minute++;

                /* Mode Data Processing */
                processModes(z, weight, &Cgroup);

                /* Calculate delays, shifts, ect */
                for (k=1; k<=CHANNELS; k++)
                {
                   delay[k]=x[k][Minute]/Cgroup +              /* Time
delay   */
                            (float)(k-1)*SAMPLE_DELAY/1000.;       /*
Sampling delay */
                   shift[k]=(int)(delay[k]*(float)F_SAMPLE);   /* # of
samples */
                                                               /*
fraction of 1 sample */
                   delta[k]=delay[k]*(float)F_SAMPLE-(float)shift[k];
                } /* for */
             } /* if (Need new shift delays)  */

             for (j=1; j<=CHANNELS; j++)
             {
                if (delay[j]>=0.0)
                {
                   arg=(float)(ORDER+1)/2.0+(1-delta[j]);
```

```
                    samples = &inSOUND[j][i-shift[j]-(ORDER+1)/2];
                  } /* if */
                  else if (delay[j]<0.0)
                  {
                     arg=(float)(ORDER+1)/2.0+delta[j];
                     samples = &inSOUND[j][i-shift[j]-(ORDER+1)/2+1];
                  } /* else if */
                 INTERPOLATE(indx, samples, ORDER+1, arg, &ans, &err);
                 buffSOUND[j][i-F_SAMPLE]=ans;
                 if (ERROR_ESTIMATE==ON)

MeanSqError[i-F_SAMPLE]=MeanSqError[i-F_SAMPLE]+err*err;
               } /* for */
               if (ERROR_ESTIMATE==ON)

MeanSqError[i-F_SAMPLE]=MeanSqError[i-F_SAMPLE]/(float)CHANNELS;
            } /* for */
         } /* else */

/* take fft multiply by weights & sum in frequency domain */

 df=(float)F_SAMPLE/(float)FFT_LENGTH;
   if (VERBOSE) printf("\nTaking FFT\n");
   for(i=1; i <=2*FFT_LENGTH; i++) sumXf[i]=0;

   for(j=1;j<=CHANNELS;j++)
   {
     for(i=1;i<=FFT_LENGTH;i++)
     {
        if(i <= FFT_TIME*F_SAMPLE)
        {
           Xf[2*i-1]=buffSOUND[j][i];
           Xf[2*i]=0;
        }
        else
        {
           Xf[2*i-1]=0;
           Xf[2*i]=0;
        }
     } /* for i */
     four1(Xf,FFT_LENGTH,1);
     k=1;

     for(i=1; i <=FFT_LENGTH-1; i+=2)
     {
        freq=df*(float)(i-1)/2;
        if(freq>=fmin && freq<=fmax)
        {
           /* positive frequencies */
           sumXf[i]=sumXf[i]+Xf[i]*weight[j][k];       /* real part */
           sumXf[i+1]=sumXf[i+1]+Xf[i+1]*weight[j][k];    /* imag part */
           /* negative frequencies */

sumXf[2*FFT_LENGTH-i]=sumXf[2*FFT_LENGTH-i]+Xf[2*FFT_LENGTH-i]*weight[j]
[k];

sumXf[2*FFT_LENGTH-i+1]=sumXf[2*FFT_LENGTH-i+1]+Xf[2*FFT_LENGTH-i+1]*wei
ght[j][k];
           k++;
        }   /* if (frequency within Bandwidth) */
     } /* for i */
```

```
    } /* for j */
    if (VERBOSE) printf("\nTaking invFFT\n");
    four1(sumXf,FFT_LENGTH,-1);

    if(firstBuffer) j=F_SAMPLE+1;
    else j = 1;
    for(i=1; i <=F_SAMPLE*FFT_TIME;i++)
    {
       outSOUND[j]=(1/(float)FFT_LENGTH)*sumXf[2*i-1];

       if(fabs(outSOUND[j])<Min) Min=fabs(outSOUND[j]);
       if(fabs(outSOUND[j])>Max) Max=fabs(outSOUND[j]);
       j++;

    } /* for i */

    /* clear summer for next set of data */
    for(i=1; i <=2*FFT_LENGTH; i++) sumXf[i]=0;

          if (VERBOSE) printf("\nWriting Data to file\n");
          if (SIGNAL==ON) putOutput();
          firstBuffer=0;  /* set firstBuffer to false */
       } /* for */

       if (VERBOSE==ON)
          printf("\t%i minutes of input data processed.\n", Minute);

       if (VERBOSE==OFF && Minute==1)
       {
          printf("\n\nAll user input has been accepted.\n");
          printf("Program is in SILENT mode.\n");
          printf("If desired, put this job in background now.\n");
       } /* if */

    } /* while not next minute*/


    if (VERBOSE==ON)
    {
       printf("EXECUTION COMPLETE: End of tilt data encountered\n");
       printf("Maximum magnitude encountered was: %e\n",Max);
       printf("Minumum magnitude encountered was: %e\n",Min);
    }
    fclose(fpInSound);
    if(SIGNAL==ON) fclose(fpOutSound);
    if(ERROR_ESTIMATE==ON) fclose(fpMSE);
    exit(0);
}/*****************************************************************/
 /************************** END main ****************************/
 /*****************************************************************/
/*****
* FUNCTION: getInput()
*
*  This function handles all acoustic input.  It also provides one of
*  two normal process terminations available in the program.  (The other
*  is located in main().)
*
*  Arguments:                    none
*
*  Return value:                 0
*
```

```
*  Functions called:                vector()          ExitOnError()
*
*  Definitions called:              ANSI              UNIX
*                                   F_SAMPLE          CHANNELS
*                                   BUFFER_TIME       VERBOSE
*                                   ERROR_ESTIMATE    SIGNAL
*
*  Global variables called:         inSOUND[][]        Min
*                                   firstBuffer        Max
*                                   fpInSound          fpOutSound
*                                   fpOutSpectrum      fpMSE
*
*  Significant memory allocation:   diskBuffer[]
*
*****/
#if defined ( ANSI )
int getInput(void)
#elif defined ( UNIX )
getInput()
#endif
{
   int i, j, buffer, items;
   float *diskBuffer;
   char fileName[80];

   if (firstBuffer)
   {
      printf("Enter file name for input acoustic data: ");

      if((scanf("%s", fileName))==EOF)
         ExitOnError("Fatal error in scanf()");

      printf("\n\n");

      if ((fpInSound=fopen(fileName, "rb")) == NULL)
         ExitOnError("Error opening INPUT ACOUSTIC data file.");
   } /* if */

   /* Memory Allocation */
   if(firstBuffer) buffer=(2+FFT_TIME)*F_SAMPLE*CHANNELS;
   else buffer=(FFT_TIME)*F_SAMPLE*CHANNELS;
   diskBuffer=(float*)vector(buffer);

   items=fread((char*)(diskBuffer+1), sizeof(float), buffer, fpInSound);
   if (VERBOSE) printf("items = %i\n",items);
   if (items==buffer) /*continue*/;
   else if(ferror(fpInSound) != 0)
      ExitOnError("Error encountered while reading input acoustic data");
   else if(feof(fpInSound) != 0)
   {
      if (VERBOSE==ON)
      {

printf("\n\t*************************************************\n");
         printf("\t    End of File reached: EXECUTION COMPLETE\n");
         printf("\t\t%i minutes of data processed\n",Minute-1);
         printf("\t\t%i bytes of data discarded\n", items*sizeof(float));
         printf("\t\tMaximum magnitude encountered was: %e\n",Max);
         printf("\t\tMinimum magnitude encountered was: %e\n",Min);
         printf("\t    End of File reached: EXECUTION COMPLETE\n");
         printf("\t*************************************************\n");
```

```
      }

      fclose(fpInSound);
      if (SIGNAL==ON) fclose(fpOutSound);
      if (ERROR_ESTIMATE==ON) fclose(fpMSE);
      exit(0);
   } /* else if */
   else ExitOnError("Unknown error handling acoustic input file.");

   for (i=1; i<=(2+FFT_TIME)*F_SAMPLE; i++)
   {
      for (j=1; j<=CHANNELS; j++)
      {
         if (firstBuffer)
         {
            inSOUND[j][i] = diskBuffer[CHANNELS*(i-1)+j];
         } /* if */
         else
         {
            if(i<=2*F_SAMPLE)

inSOUND[j][i]=inSOUND[j][i+(int)((float)FFT_TIME*(float)F_SAMPLE)];
            else
               inSOUND[j][i]=diskBuffer[CHANNELS*(i-2*F_SAMPLE-1)+j];
         } /* else */
      } /* for */
   } /* for */

   /* Deallocate Memory */
   free((char*)diskBuffer);
   return( 0 );
} /************************/
  /***** END getInput *****/
  /************************/
/*****
* FUNCTION: putOutput()
*
*  This function handles all acoustic output.  Additionally, it outputs
*  the estimated mean squared error from the interpolators (if enabled).
*
*  Arguments:                       none
*
*  Return value:                    0
*
*  Functions called:                ExitOnError()
*
*  Definitions called:              ANSI              UNIX
*                                   F_SAMPLE          BUFFER_TIME
*                                   ERROR_ESTIMATE    ASCII
*                                   BINARY
*
*  Global variables called:         outSOUND[]        MeanSqError[]
*                                   firstBuffer       fpOutSound
*                                   fpMSE
*
*  Significant memory allocation:   none
*
*****/
#if defined ( ANSI )
int putOutput(void)
#elif defined ( UNIX )
```

```
putOutput()
#endif
{
   int i, cut;
   char fileName[80], *mode;

   if(firstBuffer)
   {
      if (ASCII==ON) mode="wt";
      else if (BINARY==ON) mode="wb";

      cut=1;
      printf("Enter file name for output data: ");

      if((scanf("%s", fileName))==EOF)
         ExitOnError("Fatal error in scanf()");

      printf("\n\n");

      if ((fpOutSound=fopen(fileName, mode)) == NULL)
         ExitOnError("Error opening OUTPUT data file.");

      if (ERROR_ESTIMATE==ON)
      {
         printf("\nEnter file name for interpolator error estimate: ");
         if((scanf("%s",fileName))==EOF)
            ExitOnError("Fatal error in scanf");

         if ((fpMSE=fopen(fileName, mode)) == NULL)
            ExitOnError("Error opening error.dat");
      } /* if */
   } /* if */
   else cut=2;

   if (ERROR_ESTIMATE==ON)
   {
      if (ASCII==ON)
      {
         for (i=1; i<=(2+FFT_TIME-cut)*F_SAMPLE; i++)
            fprintf(fpMSE,"%f\n", MeanSqError[i]);
      } /* if */
      else if (BINARY==ON)
      {

if(fwrite((char*)(MeanSqError+1),sizeof(float),(2+FFT_TIME-cut)*F_SAMPLE,
            fpMSE)==(unsigned)(2+FFT_TIME-cut)*F_SAMPLE) ;
         else if(ferror(fpMSE) != 0)
            ExitOnError("Error encountered writing error data");
         else ExitOnError("Unknown error handling error file.");
      } /* else if */
   } /* if */

   if (ASCII==ON)
   {
      for (i=1;i<=(2+FFT_TIME-cut)*F_SAMPLE; i++)
         fprintf(fpOutSound, "%g\n", outSOUND[i]);
   } /* if */
   else if (BINARY==ON)
   {

if(fwrite((char*)(outSOUND+1),sizeof(float),(2+FFT_TIME-cut)*F_SAMPLE,
```

```
         fpOutSound)==(unsigned)(2+FFT_TIME-cut)*F_SAMPLE) ;
      else if(ferror(fpOutSound) != 0)
         ExitOnError("Error encountered writing output acoustic data");
      else ExitOnError("Unknown error handling acoustic output file.");
   } /* else if */
   return( 0 );
} /************************/
  /***** END putOutput *****/
  /************************/
/*****
* FUNCTION: processTilt()
*
*  This function handles all array tilt data.  It calculates the X, Y, Z
*  coordinates of each hydrophone as a function of time.  The coordinate
*  system is oriented such that X points toward the signal "origin" and
*  Z points down.
*
*  Arguments:                      x[][]          y[][]
*                                  z[][]
*
*  Return value:                   0
*
*  Functions called:               ExitOnError()  matrix()
*                                  vector()       free_matrix()
*
*  Definitions called:             ANSI           UNIX
*                                  DELTA_R        LOWER_SENSOR
*                                  RADIAN         CHANNELS
*                                  LOOK_DIRECTION OFFSET
*                                  TILT_BUFFER
*
*  Global variables called:        LastDelay
*
*  Significant memory allocation:  xx[][]         yy[][]
*                                  zz[][]         tilt[]
*                                  angle[]        udepth[]
*                                  ldepth[]       delayClock[]
*
*****/
#if defined ( ANSI )
int processTilt(float **x, float **y, float **z, float *delayClock)
#elif defined ( UNIX )
processTilt(x,y,z,delayClock)
float **x, **y, **z, *delayClock;
#endif
{
   int i, j, notEOF;
   float *tilt, *angle, *udepth, *ldepth, **xx, **yy, **zz, theta;
   char fileName[80];
   FILE *fp1, *fp2;

   /* Open Data Files */
   printf("Enter file name for upper tilt sensor data: ");

   if((scanf("%s", fileName))==EOF)
      ExitOnError("Fatal error in scanf()");

   printf("\n\n");

   if ((fp1=fopen(fileName, "rt")) == NULL)
      ExitOnError("Error opening UPPER TILT data file.");
```

```
   if (LOWER_SENSOR==ON)
   {
      printf("Enter file name for lower tilt sensor data: ");

      if((scanf("%s", fileName))==EOF)
         ExitOnError("Fatal error in scanf()");

      printf("\n\n");

      if ((fp2=fopen(fileName, "rt")) == NULL)
         ExitOnError("Error opening LOWER TILT data file.");
      ldepth=(float*)vector(TILT_BUFFER);
   } /* if */

   /* Memory Allocation */
   tilt=(float*)vector(TILT_BUFFER);
   angle=(float*)vector(TILT_BUFFER);
   udepth=(float*)vector(TILT_BUFFER);
   xx=(float**)matrix(CHANNELS, TILT_BUFFER);
   yy=(float**)matrix(CHANNELS, TILT_BUFFER);
   zz=(float**)matrix(CHANNELS, TILT_BUFFER);

   i=1;                    /* read upper tilt data */
   notEOF=1;
   while(notEOF)
   {
      if(fscanf(fp1,"%g %g %g\n",&tilt[i],&angle[i],&udepth[i]) != EOF)
i++;
      else notEOF=0;
   }

   if (LOWER_SENSOR==ON)
   {
      j=1;                    /* read lower tilt data */
      notEOF=1;
      while(notEOF)
      {
         if(fscanf(fp2, "%g\n",&ldepth[j]) != EOF) j++;
         else notEOF=0;
      } /* while */
      if (i<=j) LastDelay=i-1;
      else LastDelay=j-1;
   }
   else LastDelay=i-1;


/***  Set Delay Clock Times in Seconds  ***/

   delayClock[0]=0.0;
   for (i=1; i<=LastDelay; i++) delayClock[i]=(float)i*60.0;   /* Delay
Execution Times */

   /************This is the assumed array geometry: LINEAR**********/
   for (j=1; j<=CHANNELS; j++)
   {
      for (i=1; i<=LastDelay; i++)
      {
         xx[j][i]=DELTA_R*(float)(j-1)*cos(angle[i]/RADIAN)*
            sin(tilt[i]/RADIAN);
         yy[j][i]=DELTA_R*(float)(j-1)*sin(angle[i]/RADIAN)*
            sin(tilt[i]/RADIAN);
```

```
            zz[j][i]=DELTA_R*(float)(j-1)*cos(tilt[i]/RADIAN)+
               OFFSET*cos(tilt[i]/RADIAN)+udepth[i];
         } /* for */
      } /* for */
      /*****************************************************************/

      theta=(360.0-LOOK_DIRECTION)/RADIAN; /* coordinate rotation  */
      for (j=1; j<=CHANNELS; j++)
      {
         for (i=1; i<=LastDelay; i++)    /* points x into signal */
         {
            x[j][i]=xx[j][i]*cos(theta)-yy[j][i]*sin(theta);
            y[j][i]=xx[j][i]*sin(theta)+yy[j][i]*cos(theta);
            z[j][i]=zz[j][i];
         } /* for */
      } /* for */

      /* Memory Deallocation */
      if (LOWER_SENSOR==ON) free((char*)ldepth);
      free((char*)tilt);

      free((char*)angle);
      free((char*)udepth);
      free_matrix(xx, CHANNELS);
      free_matrix(yy, CHANNELS);
      free_matrix(zz, CHANNELS);
      fclose(fp1);
      if (LOWER_SENSOR==ON) fclose(fp2);
      return( 0 );
   } /***************************/
     /***** END processTilt *****/
     /***************************/
/*****
* FUNCTION: processModes()
*
*  This function handles the normal mode data.  It calculates hydrophone
*  weights and group speed.  The user must insure that the depth vector
*  and eigenfunction vector are of equal length.
*
*  Arguments:                    z[][]              weight[][]
*                                ptrC
*
*  Return value:                 0
*
*  Functions called:             vector()           ExitOnError()
*                                INTERPOLATE()      dydx()
*
*  Definitions called:           ANSI               UNIX
*                                PI                 SSP_LENGTH
*                                EIGVAL_LENGTH      ORDER
*                                CHANNELS           F_CARRIER
*
*  Global variables called:      Minute
*
*  Significant memory allocation: depth[]           Zn[]
*                                OnOff[]            w[]
*                                Kr[]               dwdK
*
*****/
#if defined ( ANSI )
int processModes(float **z, float **weight, float *ptrC)
```

```
#elif defined ( UNIX )
processModes(z,weight,ptrC)
float **z, **weight, *ptrC;
#endif
{
   int i, j, k, ptsEigVal, set, notEOF, deadPhones, weightNotAssigned;
   int items, moreData=ON;
   static int ptsDepth, ptsEigFun;
   float  *w, *Kr, *dwdK, *Work, err, df, freq;
   static float depth[SSP_LENGTH+1], Zn[SSP_LENGTH+1][EIGVAL_LENGTH+1],
OnOff[CHANNELS+1];
   char key, fileName[80];
   FILE *fp1, *fp2;

   if(firstBuffer==1)
   {
      w=(float*)vector(EIGVAL_LENGTH);
      Kr=(float*)vector(EIGVAL_LENGTH);
      dwdK=(float*)vector(EIGVAL_LENGTH);
      Work=(float*)vector(EIGVAL_LENGTH);

      printf("Enter file name for normal mode data (eigenfunction): ");

      if((scanf("%s", fileName))==EOF)
         ExitOnError("Fatal error in scanf()");

      printf("\n\n");

      if ((fp1=fopen(fileName, "rb")) == NULL)
         ExitOnError("Error    opening    NORMAL    MODE    data    file
(eigenfunction).");

      printf("Enter file name for normal mode data (eigenvalues): ");

      if((scanf("%s", fileName))==EOF)
         ExitOnError("Fatal error in scanf()");

      printf("\n\n");

      if ((fp2=fopen(fileName, "rt")) == NULL)
         ExitOnError("Error    opening    NORMAL    MODE    data    file
(eigenvalues).");

      i=1;                    /* read normal mode eigen values */
      notEOF=1;
      while(notEOF)
      {
         if(fscanf(fp2, "%g %g \n", &w[i], &Kr[i]) != EOF) i++;
         else notEOF=0;
      }
      ptsEigVal=i-1;

                           /* read normal mode data Zn */
      set=(ORDER+1)/2;
      j=1;
      df=2*PI*(float)F_SAMPLE/(float)FFT_LENGTH;
      while(moreData)
      {
        items=fread(Work,sizeof(float),ptsEigVal+1,fp1);
        if(items==ptsEigVal+1) /* continue */;
        else if(ferror(fp1) !=0)
```

```
            ExitOnError("Error encountered while reading Eigenfunction
File");
         else if(feof(fp1) !=0) moreData=OFF;  /* EOF */
         else ExitOnError("Unknown error handling Eigenfunction data");

         depth[j]=Work[0];
         /* perform "quick" 2nd order polynomial interpolation between
frequencies */
         freq=df;
         k=1;
         i=1;
         set=0;
         while(freq<=w[ptsEigVal])
         {
            if(freq>=w[1] && ((float)i/((float)k+2) < (w[k+1]-w[k])/df ||
k+2 == ptsEigVal))
            {

fastpoly2(&w[k-set],&Work[k-set],-(ORDER+1),freq,&Zn[j][i],&err);
               i++;
               freq+=df;
            } /* if */
            else if(w[k+1]<freq) k++;
            else freq+=df;
         }  /* while frequency in range */
         j++;
      }
         /* while(moreData) */

      fmin=w[1]/(2*PI);
      fmax=w[ptsEigVal]/(2*PI);
      printf("\nProcessing Bandwidth is from %g to %g Hz.\n\n",fmin,fmax);
      ptsEigFun=i-1;
      ptsDepth=j-1;

      for(i=1;i<=ptsDepth;i++) depth[i]=depth[i]+CTD_OFFSET;

      for (i=1;i<=CHANNELS;i++) OnOff[i]=1.0;

      printf("Do you want to turn off any hydrophones? ");

      if((scanf("%s",&key))==EOF)
         ExitOnError("Fatal error in scanf()");

      if(key=='y' || key=='Y')
      {
         printf("\nHow many hydrophones must be secured? ");

         if((scanf("%i", &deadPhones))==EOF)
            ExitOnError("Fatal error in scanf()");

         for(i=1;i<=deadPhones;i++)
         {
            printf("\nEnter hydrophone number to secure:  ");

            if((scanf("%i", &j))==EOF)
               ExitOnError("Fatal error in scanf()");

            if (j>CHANNELS || j<1)
               ExitOnError("Bad hydrophone identification");
            OnOff[j]=0.0;
```

```
      } /* for */
   } /* if */
} /* if */

for(i=1;i<=LastDelay;i++)
{
   if(depth[ptsDepth]<z[CHANNELS][i])
   {
      printf("Max eigenfunction depth is: %f\n",depth[ptsDepth]);
      printf("at depth[] index of: %i\n",ptsDepth);
      printf("Max depth of phone number %i is: %f\n\n",
          CHANNELS,z[CHANNELS][i]);
      ExitOnError("Fatal data set error");
   } /* if */
} /* for */

for (i=1; i<=CHANNELS; i++)
{
   if(OnOff[i])
   {
      for (k=1;k<=ptsEigFun;k++)
      {
         weightNotAssigned=1;
         j=1;
         while (j<=ptsDepth && weightNotAssigned)
         {
            if(z[i][Minute]<0.0 || depth[j]<0.0)
            {
               printf("i=%i\n",i);
               printf("j=%i\n",j);
               printf("Minute=%i\n",Minute);
               printf("z[i][Minute] is: %f\n", z[i][Minute]);
               printf("depth[j] is: %f\n",depth[j]);
               printf("Depth   less   than   zero   encountered   in
processModes.");
               printf("\n\n");
               ExitOnError("Check   input   depths   for   coordinate
orientation");
            } /* if */


            if(depth[j]<z[i][Minute] && depth[j+1]>z[i][Minute])
            {
               set=(ORDER+1)/2;
               INTERPOLATE(&depth[j-set],   &Zn[j-set][k],   ORDER+1,
z[i][Minute],
                  &weight[i][k], &err);
               weightNotAssigned=OFF;
            } /* if */

            if(depth[j]==z[i][Minute])
            {
               weight[i][k]=Zn[j][k];
               weightNotAssigned=OFF;
            } /* if */

            j++;
         } /* while */
      } /* for k */
   } /* if OnOff */
} /* for i */
```

```
   /* set weights to zero if phone turned off */

   for (i=1; i<=CHANNELS; i++)
   {
      if (OnOff) /* do nothing */;
      else
        for (k=1; k<=ptsEigFun; k++) weight[i][k]=OnOff[i]*weight[i][k];
   } /* for */

   if (Minute==1)
   {
      dydx(Kr,w,dwdK,ptsEigVal);
      for (i=1;i<=ptsEigVal;i++)
      {
         if (w[i]<2.0*PI*F_CARRIER && w[i+1]>2.0*PI*F_CARRIER)
         {
            set=(ORDER+1)/2;
            INTERPOLATE(&w[i-set],&dwdK[i-set],ORDER+1,
                    2.0*PI*F_CARRIER,ptrC,&err);
         } /* if */
      } /* for */

      free((char*)w);
      free((char*)Kr);
      free((char*)dwdK);
   } /* if */
   return( 0 );
} /****************************/
  /***** END processModes *****/
  /****************************/
/*****
* FUNCTION: dydx()
*
*  This function estimates derivatives.
*
*  Arguments:                    x[]              y[]
*                                ddx[]            points
*
*  Return value:                 0
*
*  Functions called:             ExitOnError()
*
*  Definitions called:           ANSI             UNIX
*                                STEP
*
*  Global variables called:      none
*
*  Significant memory allocation:  none
*
*****/
#if defined ( ANSI )
int dydx(float *x, float *y, float *ddx, int points)
#elif defined ( UNIX )
dydx(x,y,ddx,points)
float *x, *y, *ddx;
int points;
#endif
{
   int n;

   for (n=1;n<=points;n++)
```

```
   {
      if ((n>=STEP) && (n<=points-STEP))  /*center*/
         ddx[n]=(y[n+STEP]-y[n-STEP])/(x[n+STEP]-x[n-STEP]);

      else if (n<STEP)  /*beginning*/
         ddx[n]=(y[n+STEP]-y[1])/(x[n+STEP]-x[1]);

      else if (n>points-STEP)  /*end*/
         ddx[n]=(y[points]-y[n-STEP])/(x[points]-x[n-STEP]);

      else
         ExitOnError("Index error in dydx");  /* sanity check */
   } /* for */
   return( 0 );
} /*******************/
  /***** END dydx *****/
  /*******************/
/*****
* FUNCTION: fastpoly2()
*
*   This function performs second order polynomial interpolation for
interpolating
*  between modal frequency.
*
*  Arguments:                        xa[]              ya[]
*                                    n                 x
*                                    y                 dy
*
*  Return value:                     0
*
*  Functions called:                 vector()          ExitOnError()
*
*  Definitions called:               ANSI              UNIX
*
*  Global variables called:          none
*
*  Significant memory allocation:    d[]               c[]
*
*****/
#if defined ( ANSI )
int fastpoly2( float *xa, float *ya, int n, float x, float *y, float *dy)
#elif defined ( UNIX )
fastpoly2(xa,ya,n,x,y,dy)
float *xa, *ya, x, *y, *dy;
int n;
#endif
{
   int i;
   float dif, dx;

   if (abs(n) != 4)
      ExitOnError("USER error, fastpoly3 interpolater can only be ORDER
3");

   dx = xa[1]-xa[0];

   if (n == 4)     /* FIND CLOSEST POINT */
   {
      if       (x <= xa[3]+dx/2 && x > xa[3]-dx/2) i=2;
      else if (x <= xa[3]-dx/2 && x > xa[2]-dx/2) i=1;
      else if (x <= xa[2]-dx/2 && x > xa[1]-dx/2) i=0;
```

```
      else
        ExitOnError("Error in routine FASTPOLY2");
    }
    else i=1;

    dif=x-xa[i];

   *y=ya[i];
   *y-= dif*(3.*ya[i] - 4.*ya[i+1] + ya[i+2])/(2.*dx);
   *y+= dif*dif*(ya[i] - 2.*ya[i+1] + ya[i+2])/(2.*dx*dx);

   *dy=0;      /* error term to be installed later */

   return(0);

}    /*************************/
     /***** END fastpoly2 *****/
     /*************************/

/*****
* FUNCTION: fastpoly3()
*
*  This function performs Third Order polynomial interpolation.
*
*  Arguments:                        xa[]              ya[]
*                                    n                 x
*                                    y                 dy3
*
*  Return value:                     0
*
*  Functions called:                 vector()          ExitOnError()
*
*  Definitions called:               ANSI              UNIX
*
*  Global variables called:          none
*
*  Significant memory allocation:    none
*
*****/
#if defined ( ANSI )
int fastpoly3( float *xa, float *ya, int n, float x, float *y, float *dy3)
#elif defined ( UNIX )
fastpoly3(xa,ya,n,x,y,dy3)
float *xa, *ya, x, *y, *dy3;
int n;
#endif
{
    int i;
    float dif, dx, yo;

    if (abs(n) != 4)
       ExitOnError("USER error, fastpoly3 interpolater can only be ORDER
3");
    i = 1;
    dx = xa[i+1]-xa[i];
    dif = x - xa[i-1];
   yo= 4*ya[1] - 6*ya[2] + 4*ya[3] - ya[4];
   *dy3 = -ya[1] + 3.*ya[2] - 3.*ya[3] + ya[4];

   *y = yo + (dif/dx*( -13./3.*ya[1] + 9.5*ya[2] -7.*ya[3] + 11./6.*ya[4]+
             (dif/(2.*dx)*( 3.*ya[1] - 8.*ya[2] + 7.*ya[3] - 2.*ya[4] +
```

```
            (dif/(3.*dx)*( *dy3 ))))));
  return(0);

}   /*************************/
    /***** END fastpoly3 *****/
    /*************************/

/*****
* FUNCTION: polint()
*
*  This function performs polynomial interpolation.
*
*  Arguments:                       xa[]            ya[]
*                                   n               x
*                                   y               dy
*
*  Return value:                    0
*
*  Functions called:                vector()        ExitOnError()
*
*  Definitions called:              ANSI            UNIX
*
*  Global variables called:         none
*
*  Significant memory allocation:   d[]             c[]
*
*****/
#if defined ( ANSI )
int polint( float *xa, float *ya, int n, float x, float *y, float *dy)
#elif defined ( UNIX )
polint(xa,ya,n,x,y,dy)
float *xa, *ya, x, *y, *dy;
int n;
#endif
{
   int i, m, ns=1;
   float den, dif, dift, ho, hp, w;
   float *c, *d;


   n=abs(n);
   dif=fabs(x-xa[1]);

   c=(float*)vector(n);
   d=(float*)vector(n);

   for (i=1; i<=n; i++)
   {
      if ((dift=fabs(x-xa[i])) < dif)
      {
         ns=i;
         dif=dift;
      } /* if */
      c[i]=ya[i];
      d[i]=ya[i];
   } /* for */
   *y=ya[ns--];
   for (m=1; m<n; m++)
   {
      for (i=1; i<=n-m; i++)
      {
```

```
            ho=xa[i]-x;
            hp=xa[i+m]-x;
            w=c[i+1]-d[i];
           if ((den=ho-hp)==0.0)
              ExitOnError("Error in routine POLINT");
            den=w/den;
            d[i]=hp*den;
            c[i]=ho*den;
         } /* for */
         *y += (*dy=(2*ns < (n-m) ? c[ns+1] : d[ns--]));
      } /* for */
      free((char*)d);
      free((char*)c);
      return( 0 );
   } /*********************/
     /***** END polint ****/
     /*********************/
   /*****
   * FUNCTION: ratint()
   *
   *  This function performs rational function interpolation.
   *
   *  Arguments:                          xa[]              ya[]
   *                                      n                 x
   *                                      y                 dy
   *
   *  Return value:                       0
   *
   *  Functions called:                   vector()          ExitOnError()
   *
   *  Definitions called:                 ANSI              UNIX
   *                                      TINY
   *
   *  Global variables called:            none
   *
   *  Significant memory allocation:      d[]               c[]
   *
   *****/
   #if defined ( ANSI )
   int ratint( float *xa, float *ya, int n, float x, float *y, float *dy)
   #elif defined ( UNIX )
   ratint(xa,ya,n,x,y,dy)
   float *xa, *ya, x, *y, *dy;
   int n;
   #endif
   {
      int m, i, ns=1;
      float w, t, hh, h, dd, *c, *d;

      n=abs(n);
      c=(float*)vector(n);
      d=(float*)vector(n);
      hh=fabs(x-xa[1]);
      for (i=1; i<=n; i++)
      {
         h=fabs(x-xa[i]);
         if (h==0.0)
         {
            *y=ya[i];
            *dy=0.0;
            free((char*)d);
```

```
            free((char*)c);
            return ( 0 );
         } /* if */
         else if (h<hh)
         {
            ns=i;
            hh=h;
         } /* else if */
         c[i]=ya[i];
         d[i]=ya[i]+TINY;
      } /* for */
      *y=ya[ns--];
      for (m=1;m<n;m++)
      {
         for (i=1; i<=n-m;i++)
         {
            w=c[i+1]-d[i];
            h=xa[i+m]-x;
            t=(xa[i]-x)*d[i]/h;
            dd=t-c[i+1];
            if (dd==0.0)
               ExitOnError("Error in routine RATINT");
            dd=w/dd;
            d[i]=c[i+1]*dd;
            c[i]=t*dd;
         } /* for */
         *y += (*dy=(2*ns < (n-m) ? c[ns+1] : d[ns--]));
      } /* for */
      free((char*)d);
      free((char*)c);
      return( 0 );
   } /**********************/
     /***** END ratint ****/
     /**********************/
   /*****
   * FUNCTION: window()
   *
   *  This function applies a Blackman window to a vector.
   *
   *  Arguments:                          data[]            N
   *
   *  Return value:                       0
   *
   *  Functions called:                   none
   *
   *  Definitions called:                 ANSI              UNIX
   *                                      PI
   *
   *  Global variables called:            none
   *
   *  Significant memory allocation:      none
   *
   *****/

   #if defined ( ANSI )
   int window(float *data, int N);
   #elif defined ( UNIX )
   window( data, N )
   float *data;
   int N;
   #endif
```

```
{
   int n;

   for (n=0; n<N; n++)
   {

data[n+1]=data[n+1]*(0.42+0.5*cos(2.0*PI*(float)(n-N/2)/(float)(N-1))
         +0.08*cos(4.0*PI*(float)(n-N/2)/(float)(N-1)));
   } /* for */
   return ( 0 );
} /**********************/
  /***** END window *****/
  /**********************/
/*****
* FUNCTION: realft()
*
*  This function calculates FFT's
*
*  Arguments:                          data[]          n
*                                      isign
*
*  Return value:                       0
*
*  Functions called:                   four1()
*
*  Definitions called:                 ANSI            UNIX
*
*  Global variables called:            none
*
*  Significant memory allocation:      none
*
*****/
#if defined ( ANSI )
int realft(float *data, int n, int isign)
#elif defined ( UNIX )
realft(data, n, isign)
float *data;
int n, isign;
#endif
{
   int i, i1, i2, i3, i4, n2p3;
   float c1=0.5, c2, h1r, h1i, h2r, h2i;
   double wr, wi, wpr, wpi, wtemp, theta;

   theta=3.141592653589793/(double)n;
   if (isign==1)
   {
      c2 = -0.5;
      four1(data, n, 1);
   } /* if */
   else
   {
      c2=0.5;
      theta = -theta;
   } /* else */
   wtemp=sin(0.5*theta);
   wpr = -2.0*wtemp*wtemp;
   wpi=sin(theta);
   wr=1.0+wpr;
   wi=wpi;
   n2p3=2*n+3;
```

```
   for (i=2; i<=n/2; i++)
   {
      i4=1+(i3=n2p3-(i2=1+(i1=i+i-1)));
      h1r=c1*(data[i1]+data[i3]);
      h1i=c1*(data[i2]-data[i4]);
      h2r = -c2*(data[i2]+data[i4]);
      h2i=c2*(data[i1]-data[i3]);
      data[i1]=h1r+wr*h2r-wi*h2i;
      data[i2]=h1i+wr*h2i+wi*h2r;
      data[i3]=h1r-wr*h2r+wi*h2i;
      data[i4] = -h1i+wr*h2i+wi*h2r;
      wr=(wtemp=wr)*wpr-wi*wpi+wr;
      wi=wi*wpr+wtemp*wpi+wi;
   } /* for */
   if (isign==1)
   {
      data[1]=(h1r=data[1])+data[2];
      data[2]=h1r-data[2];
   } /* if */
   else
   {
      data[1]=c1*((h1r=data[1])+data[2]);
      data[2]=c1*(h1r-data[2]);
      four1(data,n,-1);
   } /* else */
   return ( 0 );
} /**********************/
  /***** end realft *****/
  /**********************/
/*****
* FUNCTION: four1()
*
*  This function calculates FFT's
*
*  Arguments:                    data[]          nn
*                                isign
*
*  Return value:                 0
*
*  Functions called:             none
*
*  Definitions called:           ANSI            UNIX
*                                SWAP
*
*  Global variables called:      none
*
*  Significant memory allocation:  none
*
*****/
#if defined ( ANSI )
int four1(float *data, int nn, int isign)
#elif defined ( UNIX )
four1(data, nn, isign)
float *data;
int nn, isign;
#endif
{
   int n, mmax, m, j, istep, i;
   double wtemp, wr, wpr, wpi, wi, theta;
   float tempr, tempi;
```

```
    n=nn << 1;
    j=1;
    for (i=1; i<n; i+=2)
    {
       if (j > i)
       {
          SWAP(data[j],data[i]);
          SWAP(data[j+1],data[i+1]);
       } /* for */
       m=n >> 1;
       while (m >= 2 && j > m)
       {
          j -= m;
          m >>=1;
       } /* while */
       j += m;
    } /* for */
    mmax=2;
    while (n > mmax)
    {
       istep=2*mmax;
       theta=6.28318530717959/(isign*mmax);
       wtemp=sin(0.5*theta);
       wpr = -2.0*wtemp*wtemp;
       wpi=sin(theta);
       wr=1.0;
       wi=0.0;
       for (m=1; m<mmax; m+=2)
       {
          for (i=m; i<=n; i+=istep)
          {
             j=i+mmax;
             tempr=wr*data[j]-wi*data[j+1];
             tempi=wr*data[j+1]+wi*data[j];
             data[j]=data[i]-tempr;
             data[j+1]=data[i+1]-tempi;
             data[i] += tempr;
             data[i+1] += tempi;
          } /* for */
          wr=(wtemp=wr)*wpr-wi*wpi+wr;
          wi=wi*wpr+wtemp*wpi+wi;
       } /* for */
       mmax=istep;
    } /* while */
    return ( 0 );
} /*********************/
  /***** end four1 *****/
  /*********************/
/*****
* FUNCTION: vector()
*
*  This function allocates memory for UNIT OFFSET vectors.
*
*  Arguments:                    length
*
*  Return value:                 *v
*
*  Functions called:             ExitOnError()
*
*  Definitions called:           ANSI              UNIX
*
```

```
*  Global variables called:         none
*
*  Significant memory allocation:   v[]
*
*****/
#if defined ( ANSI )
float *vector(int length)
#elif defined ( UNIX )
vector(length)
int length;
#endif
{
   float *v;

   if ((v=(float*)malloc((length+1)*sizeof(float)))==NULL)
      ExitOnError("Memory allocation failure in vector().");
#if defined ( ANSI )
   return v;
#elif defined ( UNIX )
   return (long int)v;
#endif
} /**********************/
  /***** END vector *****/
  /**********************/
/*****
* FUNCTION: matrix()
*
*  This function allocates memory for UNIT OFFSET 2-D arrays.
*
*  Arguments:                       row              col
*
*  Return value:                    **m
*
*  Functions called:                ExitOnError()
*
*  Definitions called:              ANSI             UNIX
*
*  Global variables called:         none
*
*  Significant memory allocation:   m[][]
*
*****/
#if defined ( ANSI )
float **matrix(int row, int col)
#elif defined ( UNIX )
matrix(row,col)
int row, col;
#endif
{
   int i;
   float **m;

   if ((m=(float**)malloc((unsigned)(row+1)*sizeof(float*)))==NULL)
      ExitOnError("Allocation failure 1 in matrix()");
   for (i=1; i<=row; i++)
   {
      if ((m[i]=(float*)malloc((unsigned)(col+1)*sizeof(float)))==NULL)
         ExitOnError("Allocation failure 2 in matrix()");
   } /* for */
#if defined ( ANSI )
   return m;
```

```
#elif defined ( UNIX )
   return (long int)m;
#endif
} /**********************/
  /***** END matrix *****/
  /**********************/
/*****
* FUNCTION: free_matrix()
*
*  This function deallocates memory from UNIT OFFSET 2-D arrays.
*
*  Arguments:                      m[][]              row
*
*  Return value:                   0
*
*  Functions called:               none
*
*  Definitions called:             ANSI           UNIX
*
*  Global variables called:        none
*
*  Significant memory allocation:  none
*
*****/
#if defined ( ANSI )
void free_matrix(float **m, int row)
#elif defined ( UNIX )
free_matrix(m,row)
float **m;
int row;
#endif
{
   int i;

   for(i=row; i>=1; i--)
      free((char*)m[i]);
   free((char*)m);
   return( 0 );
} /***************************/
  /***** END free matrix *****/
  /***************************/
/*****
* FUNCTION: ExitOnError()
*
*  This function performs an abnormal process termination.
*
*  Arguments:                      error_txt[]
*
*  Return value:                   none
*
*  Functions called:               none
*
*  Definitions called:             ANSI               UNIX
*
*  Global variables called:        none
*
*  Significant memory allocation:  none
*
*****/
#if defined ( ANSI )
void ExitOnError(char error_txt[])
```

```
#elif defined ( UNIX )
ExitOnError(error_txt)
char error_txt[];
#endif
{
   fprintf(stderr,"Program run-time error ...\n");
   fprintf(stderr,"%s\n",error_txt);
   fprintf(stderr,"...now exiting to system...\n");
   fclose(fpInSound);
   if(SIGNAL==ON) fclose(fpOutSound);
   if(ERROR_ESTIMATE==ON) fclose(fpMSE);
   exit(0);
} /***************************/
  /***** END ExitOnError *****/
  /***************************/
```

B. MODAL DECOMPOSITION PROGRAM

This program performs modal decomposition for a specified bandwidth and frequency increment. Inputs to the program include sound speed and density profiles. An example input file can be found at section D of this Appendix. This program also uses a complex eigen solver called as an external subroutine. Output from this program is a direct access binary file containing the eigen solutions evaluated at the specified frequencies.

```
* Last Updated by G.A. Omans II
*               on 30 July 1992
*
*************************************************************************
*****
* MODAL DECOMPOSITION PROGRAM
*
*    Program to calculate eigenmodes and eigen values given the sound
*       & density profile
*       Output is to a DIRECT ACCESS Binary file "modes.dabin"
*          Program used to extract Neccesary Data for Beam Forming is
"CWextract.f"
*       or "BBextract.f". CWextract creates input to the Continuous-Wave
*    beamformer. BBextract creates input files for use in the Broadband
*    beamformer.
*************************************************************************
*****
* FILES:
*    afreqmode.in - Input file containing sound & density profiles,
*                depth increment dz, hard bottom depth Hdepth,
*                center frequency fc (Not used in computations).
```

```
*               Profiles need not be complete, they will be inter-
*               polated to correct number of points.  However, the
*               profile depth spacing must be greater than dz at all
*               points.
*
*    modes.dabin  - Direct access, binary output file.  This file can be
*               accessed by "mextract.f" to extract modal information
*               for beamforming, or for inspecting mode shapes &
*               eigenvalues.
*
*    mode.sys     - This file is used as a check on how well the eigen
*               solver "eigrf" and the eigen subroutine itself are
*               doing.  A fair amount of information is output to
*               this file, including sound & density profiles, the
*               eigen matrix a, the number of trapped modes & their
*               eigenvalues; to name a few.  If VERBOSE parameter
*               option is selected, then even more information is
*               displayed.
*
*    standard output    - Information sent to standard output is very
limited.
*               The user is provided with just enough information to
*               indicate where the program is currently running, and
*               any significant problems if encountered.  If you
*               have selected VERBOSE prior to compiling the program,
*               then the amount of information will overwelmingly
*               increase, and will be impossible to keep track of in
*               the interactive mode.  In this case the program should
*               be run in the background and standard output should
*               be redirected to a file.  In any case, the program
*               has a significant run time do to the recursive eigen
*               solver routine, and should be run in the background
*               most of the time.
*************************************************************************
*****
*QUIRKS:
*    Because of the recursive operations performed by IMSL subroutine
*    eigrf, a large amount of memory is required.  To accomadate the
*    memory needs of the subroutine, in the main program the arrays Z
*    and kc have been demensioned larger than required by eigrf itself.
*    This extra cusion of memory significantly improves the solutions
*    from the eigen solver and increases the number of trapped modes found.
*************************************************************************
*****
* a        - (array) eigenvalue coefficients from difference eqn.
*            (small dz approx.)
* buff     - (array) profile buffer for input into READP subroutine
* c        - (array) sound speed, interpolated from input file
* depth    - (par) used to define the soft depth; This parameter
*               does not effect calculations, however, it is used for
*               testing the reliability of trapped eigen solutions at
*            the silt.
* df       - (par) frequency increment used in evaluating band
* dz       - (var) depth step read from input file
* efun     - (array) eigen functions
* eigen    - SUBROUTINE to calculate eigenfunctions and eigen values
* EX       - (log) switch used for testing the existance of a file
* f        - (var) frequency (Hz)
* fc       - (var) center frequency; serves no purpose except as identifier
* fdate      - (char) used for printing start and end times
* filename   - (char) holds the name of a file for error messages
```

```
* fmax    - (par) maximum frequency in band to evaluate
* fmin    - (par) minimum frequency in band to evaluate
* Hdepth     - (var) depth of channel + the sediment depth
*            ie. depth of hard bottom
* ifreq      - (var) frequency counter
* icount     - (var) number of terms read from profile for input into buff
* isize      - (parameter) size of temporary array used for interpolating
profile
*           NOTE: isize must be > IFIX(1.5 + ZMAX profile/DZ)
* irec    - (var) record number for direct access file
* itrap   - (var) passed by subroutine eigen indicates number of trap modes
found
* jump    - (par) jump is the maximum number of "Trapped" Modes DESIRED
*           jump is also used to jump to certain records in the direct
access
*              file.
* k       - (array) eigenvalues or the square of Kn for each mode
* m       - (par) number of modes to be calculated; for best results,
*              this should be isize-1
* nf         - (var) total number of frequencies to calculate nf =
1+(fmax-fmin)/df
* nmodes  - total number of eigen solutions for a particular frequency
written
*           to direct access output file. (minimum between itrap & jump)
* nz      - (var) number of eigenvalues, profile points, & eigenvalue
length
* nzp1    - (var) number of points in svp (includes surface point, which
is
*               not required for eigenvalue since the value is zero at the
surface)
* pi         - (var) pi itself in double precision
* rcl     - (var) record length of direct access file (important to
mextract.f)
* rho        - (array) density array,contained in input file mode.in
* temp       - (array) a working array used in subroutines readp and
eigen; in
*           eigen temp is called mstor
* w       - (var) radial frequency (rad/s)
* VERBOSE    - (log) Gives User full Status Report on All Eigen Solutions
*              and Profiles; Good For Trouble Shooting But Maybe
Overwhelming
*************************************************************************
*****
*    NOTE:  The input profiles are double precision.
*           Profiles are changed to double precision in READP subroutine.
*           Profiles in the input file need not be in double precision.
*************************************************************************
*****
C
C    SYSTEM PARAMETERS
C
      real*8 df,depth
      logical ON,OFF,VERBOSE
      parameter(ON=1,OFF=0)
C
C    USER PARMAMETERS
C
      parameter(VERBOSE=OFF)
      parameter(jump=20)
      parameter(fmin=208,fmax=240,df=1)
      parameter(isize=211,m=(isize-1))
```

```
      parameter(depth=276.)

      implicit real*8 (a-h,o-z)
      real*8 k(m),w,f,fc
      real*8 Hdepth,dz
      real*8 pi
      integer irec,nmodes,rcl
      character*24 fdate,filename
      logical EX
      DIMENSION c(isize),rho(isize),temp(isize)
      DIMENSION buff(isize,2)
C
C    EIGEN SUBROUTINE VARIABLES
C
      DIMENSION ea(isize,isize),work(2*(isize+2)*isize)
      DIMENSION kc(isize+40),z(isize+40,isize+40),efun(isize,m)
C
C    FILES
C
      OPEN(UNIT=1,STATUS='OLD',FILE='afreqmode.in')

C
c-------------------------------------------
c MORE open statements
c-------------------------------------------
c
c       OPEN MODE.SYS
c

      INQUIRE(FILE='mode.sys',EXIST=EX)
      IF(EX)THEN
              OPEN(UNIT=13,FILE='mode.sys',STATUS='OLD')
              CLOSE(13,STATUS='DELETE')
      ENDIF
        OPEN(UNIT=9,FILE='mode.sys',
     1       FORM='FORMATTED',STATUS='NEW',ERR=2003)

C
C    OPEN MODES.DABIN
C

      nf = 1+((fmax-fmin)/df)

      IF(m.LE.isize.AND.(nf+2).LE.isize) then
         rcl=isize*8 + 8*21
      ELSEIF((nf+2).LE.m) then
         rcl=m*8 + 8*21
      ELSE
         rcl=(nf+2)*8 + 8*21
      ENDIF
      INQUIRE(FILE='modes.dabin',EXIST=EX)
      IF(EX)THEN
              OPEN(UNIT=14,FILE='modes.dabin',STATUS='OLD')
              CLOSE(14,STATUS='DELETE')
      ENDIF
     OPEN(UNIT=4,FILE='modes.dabin',ACCESS='DIRECT',RECL=rcl,
     1       FORM='UNFORMATTED',STATUS='NEW',ERR=2005)

C
C       DECLARE PI IN DOUBLE PRECISION
C
```

```
      pi = 4.d0 * datan(1.d0)

C
C       Give START TIME
C
       write(6,*) fdate()
       write(6,*)'\n'

      WRITE(6,*)'RECORD LENGTH: ',rcl

C
C    INITIALIZE BUFFER VARIABLE
C
       do 10 i=1,isize
      do 20 j=1,isize
          buff(j,i) = 0.0
   20     continue
   10 continue
c
c
c    Read in ocean/model parameters
c
       read(1,*) Hdepth
       write(6,*) 'depth w/ sediment = ',Hdepth
       read(1,*) dz
       read(1,*) fc
       write(6,*)'dz =',dz
       write(6,*) 'frequency (Hz) = ',fc
C
C       DETERMINE NUMBER OF POINTS IN PROFILES
C
      nzp1 = 1.5 + Hdepth/dz
      nz = nzp1 - 1

c
c    Read in and interpolate sound velocity profile
c         Read Sound Velocity Profile into Buffer Variable & Print
c
      i = 0
      do 30 WHILE(buff(i,1).NE.-1 .AND.i.LT.isize)
        i = i+1
     read(1,*) (buff(i,j),j=1,2)
   30 continue
      icount = i-1
      write(6,*) 'SVP icount = ',icount
C
C         Read and interpolate SVP
C
      CALL READP(dz,nzp1,isize,c,temp,buff)
C
C         CLEAR BUFFER
C
      do i=1,isize
         buff(i,1)=0
         buff(i,2)=0
      enddo

C
C         Write the interpolated SVP to the screen
C
       if(VERBOSE) then
```

```
        do 60 j=1,(nzp1+1)
          write(6,*) dz*(j-1), c(j)
   60   continue
        endif

C
C    READ IN DENSITY PROFILE
C
        i=0
        DO 31 WHILE(buff(i,2).NE.-1 .AND.i.LT.isize)
          i = i+1
       read(1,*) (buff(i,j),j=1,2)
   31  continue
       icount = i-1
       write(6,*) 'Density icount = ',icount
C
C          Read and interpolate DENSITY
C
       CALL READP(dz,nzp1,isize,rho,temp,buff)
c
c          Write the interpolated DENSITY to the screen
c
       if(VERBOSE) then
       do 61 j=1,(nzp1+1)
          write(6,*) dz*(j-1), rho(j)
   61  continue
       endif

C
C    CLEAR WORKING ARRAY TEMP FOR USE AS MSTOR IN EIGEN SUBROUTINE
C
       do j=1,isize
          temp(j) = 0
       enddo
C
C    CALCULATE EIGEN FUNCTIONS & EIGEN VALUES
C                FOR ALL FREQUENCIES

C    ** nf = 1+((fmax-fmin)/df) ** CALCULATION PRIOR TO OPENING DATA FILE
       f = fmin
       write(6,*)'nf =',nf,' df =',df

c
c ***************************************************************

       WRITE(4,REC=1) fc,fmin,fmax,df,nf,dz,m,nzp1,jump,rcl
       WRITE(4,REC=2) c
       WRITE(4,REC=3) rho
c ***************************************************
c
        irec=4

       do 100 ifreq = 1,nf


C
C    CONVERT TO RADIAL FREQUENCY
C
          w = 2 * pi * f
          write(6,*)'\n\n*****************************************'
          write(6,*) 'f =',f,' (Hz)  ifreq =',ifreq
```

```
         write(6,*) 'w =',w,' (rad/s)'
C
C    INITIALIZE VARIABLES
C
         do i=1,nz-1
            k(i)=0
            kc(i)=0
            do j=1,nz-1
               if(j.LT.m) efun(i,j)=0
               z(i,j)=0
            enddo
         enddo
C
C    CALL EIGEN SOLVER SUBROUTINE
C

          CALL eigen(c,rho,dz,nz,efun,m,k,kc,w,z,ea,
     &               depth,work,temp,ifreq,isize,
     &               itrap,VERBOSE)


c
c ************************************************
      write(4,REC=irec) w,itrap,k
c ************************************************
c
       if(jump.GT.itrap) then
          nmodes = itrap
       else
          nmodes = jump
       endif

       do 70 im=1,(nmodes)
         irec=irec+1

c ************************************************
       write(4,REC=irec) (efun(i,im),i=1,nz)
c ************************************************

    70   continue
         irec=(jump+2)*ifreq + 4

   100   f=f+df

         go to 2500

c-------------------------------------------------
c close statements
c-------------------------------------------------
2003       filename='mode.sys'
      goto 2010
2005       filename='modes.dabin'
2010       write(6,3000) filename
         GOTO 2500

2500   close(9)
       close(4)
       close(1)
C
C    FORMAT STATEMENTS
C
```

```
3000      format(' ERROR OPENING FILE : ',A)

C
C       END TIME
C

      write(6,*)'Program End Time: ',fdate()

      stop
      end


*************************************************************************
*****
*************************************************************************
*****
c    SUBROUTINE EIGEN
*************************************************************************
*****
* INCLUDING DENSITY AND SOUND SPEED VARIATION
* NORMALIZED
* :eigenvalues,k and eigenvectors,z are complex
*
* compute m normal modes z(i,m) and wavenumbers k(i)
* for a given sound-speed profile c(i),
* a given frequency w,
* a given density profile
* and a given step size dz.
*
* nz*dz is the depth of the ocean
* bc:     pressure release surface.
*    rigid bottom with soft sediment interface
* input: nz,m,dz,w,c
* output: w
*         m,nz
*         c,k,z (freq.,sound speed,eigenvalues,eigenvectors)
*         (note: eigenfunctions,z have two less points than c;
*           recall z(0) is 0 and z(nz)=z(nz-1).
* internal: a,work; one must set iw=15*n
*
*************************************************************************
*
* UPDATES/CORRECTIONS:
*
*                  1) UPDATED - OUTPUT ONLY TRAPPED MODES BASED ON
*             K AT THE BOTTOM & ELIMINATES EVANESCENT
*             SOLUTIONS. 15 JUN 92
*              2) CORRECTION - EIGENMODE NORMALIZATION NOW REFLECTS
*             INVERSE OF THE DENSITY VS. DENSITY MULTIPLICATION.
*             18 JULY 92
*              3) UPDATED - REORDER EIGEN SOLUTIONS BY MODE NUMBER.
*             REORDERING IS ACCOMPLISHED BY SORTING BY EIGENVALUE
*             FROM HIGHEST TO LOWEST.  THIS WILL EFFECTIVELY ORDER
*             ALL MODES FROM LOWEST TO HIGHEST. 24 JULY 92
*              4) UPDATED/CORRECTED - REALIGNS TRAPPED MODES TO HAVE THE
*             SAME SIGN OR PHASE.  REFERENCE IS A NEGATIVE SIGN AT
*             THE BOTTOM. 29 JULY 92
*              5) UPDATED - CHECKS FOR BAD EIGENFUNCTION SOLUTIONS BY
*             EVALUATING THE EXISTANCE OF MODAL ENERGY AT THE BOTTOM.
*             30 JULY 92
```

```
*
*************************************************************************
* a       - finite difference matrix or "eigen matrix"
* all       - integer defining domain of eigen solutions
* c       - sound speed profile for an individual station
* cb          - value of the sound speed at the bottom of the water column
*           sediment
* d       - density profile for an individual station
* depth       - depth of soft bottom, used to test for energy at bottom
* dterm       -  a  term  used  in  the  finite  difference  equation  for
calculating
*                  the eigen matrix
* dz          - depth increment of input
* dz2         - dz squared, used in finite difference EQ
* dz2inv  - inverse of dz2
* dz2inv2     - two times the inverse of dz2
* efun        - eigenfunction matrix (Eigen Vectors) Represents Modes
* eigrf       - IMSL matrix eigenvalue solver subroutine
* ext         - Minimum value for sign testing of eigen function
* itrap       - counter used in conjunction with mstor to save only
*           trapped modes (FINAL itrap IS TOTAL TRAPPED MODES)
* ijob        - = 1: calculate k
*           = 2: calculate k & z
*           = 3: calculate k,z & performance index
* ik          - simular use as im
* im          - used as a modal incrementer
* isize       - a demensioning term
* jlast       - integer variable used to check number of modes realigned
* k       - real eigenvalue vector, k = Kn**2
* kb2         - eigenvalue at the bottom of water column sediment squared
* kc          - complex eigenvalue matrix
* k_trap  - variable used to relay trapped Kn's
* k_last  - test variable used in reordering eigen solutions
* m       - number of modes Evaluated from eigrf output
* mode.sys    - System file for subroutine, make sure you check "p index"
* mreal       - Number of Modes Realigned (CURRENTLY DOESN'T WORK)
* mstor       - array used to keep track of only modes which are trapped
*           efun are then stored as a function of mstor
* nz            - nzp1-1, number of depth points; also is eigen function
size
* nzp1        - number of physical points (including free surface)
* pi          - itself pi, in double precision
* VERBOSE     - logical switch for print out control
* w       - frequency (2*pi*f)
* work        - internal work space: require for IMSL routine "eigrf"
* z       - eigenfunction matrix output from eigrf subroutine
*************************************************************************
*************************************************************************

      SUBROUTINE eigen(c,d,dz,nz,efun,m,k,kc,w,z,a,
     &                 depth,work,mstor,ifreq,isize,
     &                 itrap,VERBOSE)

      logical ON,OFF
      parameter(ON=1,OFF=0)
      implicit real*8 (a-h,o-z)
      real*8 k(m),k_trap,k_last,kb2,pi,depth
      integer i,j,ik,jlast,im,itrap,all
      complex*16 kc(nz),z(nz,nz)
      dimension c(isize),d(isize),a(nz,nz),work(2*(2+isize)*isize)
      dimension efun(isize,m)
```

```
      dimension mstor(isize)
      logical VERBOSE,FLAG
      data ijob/2/

C
C    INITIALIZE ARRAY A
C
      DO 10 i=1,nz
     DO 20 j=1,nz
         a(i,j) = 0.d0
   20    CONTINUE
   10 CONTINUE
C
C    INITIALIZE WORKING ARRAY
C
      DO i=1,((2+isize)*isize*2)
         work(i) = 0.0
      ENDDO
C
C    DEFINE COMPUTATIONAL DOMAIN
C
      nzp1 = nz +1

C
C    PRINT HEADER
C
      if(ifreq.eq.1) then

      write(9,*)'NMODE PROGRAM INITIATED'
      write(9,*)'TOTAL MODES EVALUATED m = ',m
      write(9,*)' dz(m)=',dz,'  nz=',nz
      write(9,*)
      write(9,*)'sample input sound speed profile at ir=1,itr=1'
      write(9,'(4(i4,1x,f8.3))') (iz,c(iz),iz=1,nzp1,2)
      write(9,*)'sample input density profile at ir=1,itr=1'
      write(9,'(4(i4,1x,f8.3))') (iz,d(iz),iz=1,nzp1,2)
      endif

C
C    DEFINE PI

      pi = 4.d0 * datan(1.d0)

c INPUT geometrics parameters
c
c frequency is in hertz*2*pi=radians/sec
C
c dz given in m
C
      dz2inv = 1.d0/dz**2
      dz2inv2 = 2.d0*dz2inv
      dz2 = dz*2.d0

         write(6,*)'CALCULATING A MATRIX'

         do 11 iz=2,nzp1

c set up the operator matrix a in band symmetric mode
         if (iz.lt.nzp1) then
      dterm = (dlog(d(iz+1))-dlog(d(iz-1)))/dz2
```

```
      else
      dterm = 0.d0
      endif
         if (iz.lt.nzp1) a(iz-1,iz) = dz2inv-dterm
         if (iz.gt.2) a(iz-1,iz-2) = dz2inv+dterm
   11    a(iz-1,iz-1) = (w/c(iz))**2 - dz2inv2
c soft upper b.c. incorporated above
c hard lower b.c.

      a(nz,nz-1) = dz2inv2
C
C    FOR FIRST FREQUENCY OR IF VERBOSE ON, OUTPUT EIGEN MATRIX
C

      if(ifreq.eq.1 .OR. VERBOSE) then

      write(9,*)'\n check input array A(1,1 ........ 1,5)'
      write(9,*)'                          :            :'
      write(9,*)'                          :            :'
      write(9,*)'                          :            :'
      write(9,*)'                     (5,1 ........ 5,5)\n'

      do i=1,5
      write(9,*) (a(i,j),j=1,5)
      enddo
      write(9,*)'\n check input array A(nz-5,nz-5..... n-5,n)'
      write(9,*)'                          :            :'
      write(9,*)'                          :            :'
      write(9,*)'                          :            :'
      write(9,*)'                     (nz,nz-5........ nz,nz)\n'
      do i=(nz-5),nz
      write(9,*) (a(i,j),j=(nz-5),nz)
      enddo
      endif

c
c find the k*k's and the z's
c
c ijob = 1, calculate k only
c ijob = 2, calculate both k and z
c ijob = 3, calculate k, z, and performance index

      write(6,*)'SOLVING FOR EIGEN VALUES & FUNCTIONS'
      CALL eigrf(a,nz,nz,ijob,kc,z,nz,work,ier)

      write(9,*)'\n ifreq = ',ifreq,' w = ',w
      write(9,*) '\n p index =',work(1)

      if(work(1).ge.50.) then
        write(6,*)'\n\nWARNING WARNING WARNING WARNING'
        write(6,*)'P INDEX',work(1),' IS TOO LARGE'
        write(9,*)'\n\nWARNING WARNING WARNING WARNING'
        write(9,*)'P INDEX',work(1),' IS TOO LARGE'
      endif
C
C    YOU DON'T REALLY WANT TO LOOK AT THIS, IT'S BORING
C
C      if(VERBOSE) then
C         write(9,*)'complex first eigenvector'
C         do iz=1,nz,2
C            write(9,*) iz,z(iz,1)
```

```
C         enddo
C       endif

C
C    TEST K'S FOR TRAPPED MODES
C

C    DTERMINE K AT THE BOTTOM

      write(6,*)'DETERMINING TRAPPED MODES'

      cb = 1677.0
      kb2 = (w/cb)**2
      itrap = 0
      write(9,*)'\n MODE FREQ:',w/(2*pi),'\n'

C    DEFINE DOMAIN OF EIGEN SOLUTIONS

         if(nz.lt.m) then
            all=nz
         else
            all=m
         endif

C    TEST FOR TRAPPED MODES

         do 14 im=1,all
         IF(abs(dimag(kc(im))).GT.0.d0 .OR.
     &       dreal(kc(im)).LT.kb2) then
           IF(VERBOSE) THEN
                WRITE(*,*) 'MODE at im = ',im,' AT FREQ',w/(2*pi),
     &               ' Hz IS NOT TRAPPED'
                IF(abs(dimag(kc(im))).GT.0.d0) WRITE(*,*)
     &                'K^2 IMAG =',dimag(kc(im))
           IF(dreal(kc(im)).LT.0.d0) WRITE(*,*)
     &               'K^2 (NEG) =',dreal(kc(im))
           IF(dreal(kc(im)).LT.kb2 .AND. dreal(kc(im)).GT.0.d0) WRITE(*,*)
     &             'K^2 REAL < KB^2 =', dreal(kc(im)),' <',kb2
           ENDIF
         ELSE
            itrap = itrap + 1
            mstor(itrap) = im
            k_trap = dreal(kc(im))
            write(9,*)'Mode Evaluated at im =',im,
     &                      ' is TRAPPED'
            write(9,*)'eigen value:  Kn =',sqrt(k_trap)
         ENDIF

   14    continue

c ************************************************
      WRITE(9,*) 'NUMBER OF MODES \n',itrap
c ************************************************

      if(ipram.eq.1) goto 330

C
C    NORMALIZE EIGENFUNCTION (normalize the z's, store in efun)
C
```

```
          im=1
          mreal=0
          jlast=0

          do 13 WHILE(im.LE.itrap)
          k_trap=0

          IF(im.EQ.1) THEN
             do ik=1,itrap
                if(dreal(kc(mstor(ik))).GT.k_trap) then
                   k_trap=dreal(kc(mstor(ik)))
                   j=ik
                endif
             enddo
          ELSE
             do ik=1,itrap
                if(dreal(kc(mstor(ik))).GT.k_trap.AND.
     &             dreal(kc(mstor(ik))).LT.k_last) then
                   k_trap=dreal(kc(mstor(ik)))
                   j=ik
                endif
             enddo
          ENDIF

C    COUNT REALIGNED MODES

          if(j.NE.(jlast+1)) mreal = mreal+1

          k(im) = k_trap
          k_last = k_trap
          orthnorm = 0.0

             do i=1,nz
             orthnorm =orthnorm +
     &         (1.0/d(i+1))*(dreal(z(i,mstor(j)))**2 +
     &          dimag(z(i,mstor(j)))**2)
             enddo

          orthnorm =dsqrt(orthnorm*dz)

C   TAKE REAL PART AND STORE IN EFUN ARRAY

             do i=1,nz
             efun(i,im) = dreal(z(i,mstor(j)))/orthnorm
             enddo
          jlast=j
   13     im = im + 1

       write(9,999) mreal
  999          format(i4,' modal realignments made.')



c
c eigenfunction alignment (refr.radial dependency)
c method:test for significant value (>0)
c        test for sign change in gradient
c        save sign of first extremum
c
c     set up standard of comparison
```

```
      mreal=0
      FLAG=ON

      DO 250 im=1,itrap
      ext = .002
      iz = nz

      do 251 while(dabs(efun(iz,im)).LT.ext.AND.iz.GT.1)
  251 iz = iz-1

C     TEST FOR ZERO ENERGY AT BOTTOM (FAULTY EIGENFUNCTIONS)

      if(dfloat(efun(ifix(depth/float(dz)),im)).EQ.0.) then
         if(FLAG) then
            write(*,*)'\nWARNING! WARNING!\n'
            write(9,*)'\nWARNING! WARNING!\n'
            FLAG=OFF
         endif
         write(*,1002) im
         write(9,1002) im
      endif

C     ALIGN ALL EIGENFUNCTIONS TO BE NEGATIVE AT THE BOTTOM

      if(dsign(1.d0,dreal(efun(iz,im))).GT.0) then
         mreal=mreal+1
         do j=1,nz
            efun(j,im) = -efun(j,im)
         enddo
      endif

  250 CONTINUE

            write(9,1001) mreal
 1001       format(i4,' modal sign changes made.')
 1002       format('\tMode',i3,' May Have a Faulty Eigen Function')
C
C     OUTPUT SAMPLE OF EIGEN FUNCTIONS TO MODE.SYS
C              IF VERBOSE SET TO ON
C
      if(VERBOSE) then
         j = 1
         write(9,1003)j,k(j),(efun(iz,j),iz=1,nz)
         j = 20
         write(9,1003)j,k(j),(efun(iz,j),iz=1,nz)
         j =25
         write(9,1003)j,k(j),(efun(iz,j),iz=1,nz)
      endif

1003  format('    mode',i,' eval=',d11.5/' efun:',
     &       5(2x,d11.5)/22(6x,5(2x,d11.5)/) )


 330  write(6,*) '\n FINISHED CALCULATING EIGENMODES'
      return
      end

*****************************************************************
C     Subroutine READP
C
C      PROFILE READER. INTERPOLATES TO FILL IN PROFILES VALUES BETWEEN
```

```
C     INPUTS.
C
C    DZ - VERTICAL STEP SIZE
C    M  - NUMBER OF POINTS IN VERTICAL ARRAY
C    N  - NUMBER OF POINTS IN WORKING ARRAY
C         MUST BE > OR = ZMAX(PROFILE)/DZ + 1.5
C    AF - FINAL VALUES OF INTERPOLATED PROFILE ARRAY
C    A  - WORKING ARRAY OF VALUES
C         NOTE: ACTUAL PROFILE VALUES MUST BE > -100.0
C               OR INITALIZATION VALUES MUST BE CHANGED
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
      SUBROUTINE READP(DZ,M,N,AF,A,IN)
      REAL*8 A,DZ,AI,ZI,AF,IN
      INTEGER k
      DIMENSION A(N),AF(M),IN(N,2)
      ILAST = -1
C
C    initialize output array
C
      DO 7 I=1,M
     AF(I) = 0.0
    7 CONTINUE
C
C    INITIALIZE WORKING ARRAY
C         PROFILE VALUE MUST BE LESS THAN INITALIZATION VALUE
C
      DO 1 I=1,N
         A(I)=-100.0
    1 CONTINUE

      k=1
      ZI=IN(k,1)
      AI=IN(k,2)
      A(1)=AI
      I=1.5+ZI/DZ
      A(I)=AI

    2 IF(ILAST .LT. 0) THEN
         k = k+1
         ZI=IN(k,1)
         AI=IN(k,2)
      ENDIF

      IF(ZI .GE. (M-1)*DZ) ILAST=1.5+ZI/DZ
C
C         EXIT LOOP IF DONE READING PROFILE
C             PROFILE MORE THAN ONE PAST THE BOTTOM
C             OR HAS REACHED THE LAST POINT INPUTED
C
          IF(ZI.LT.0.0) GO TO 3
          I=1.5+ZI/DZ
C
C         ENSURE THAT N HAS BEEN DEFINED LARGE ENOUGH
C
          IF (I .GT. N) THEN
          WRITE(*,*) 'N DEFINED TOO SMALL: N MUST BE > NZ+2',
     &                  ' IN INPUT PROFILE, N = ',N,' I = ',I
     ELSE
             A(I)=AI
     ENDIF
      IF(ILAST.GE.0) GO TO 3
```

```
      GO TO 2
C
C     SET A(M+2) EQUAL TO THE LAST VALUE IN THE PROFILE
C          VALUES AT DEPTHS BELOW THE LAST DEFINED POINT
C          ARE GIVEN THE SAME VALUE AS THE LAST DEFINED POINT
C          UNLESS A POINT DEEPER THAN THE BOTTOM IS GIVEN
C
    3 IF(ILAST .LT. 0) A(M+2)=A(I)
      I=1
      J=1
    4 I=I+1
C
C     FIND NEXT DEFINED PROFILE VALUE (FIRST VALUE ALREADY GIVEN)
C          NOTE: PROFILE VALUES MUST BE LESS THAN INITIALIZATION VALUE
C
      IF(A(I).LE.-100.0)GO TO 4
C
C     IF TWO CONSECUTIVE PROFILE POINTS ARE GIVEN, NO INTERPOLATION IS
NEEDED
C
      IF(I-J.EQ.1)GO TO 6
C
C     INTERPOLATE IN BETWEEN THE TWO DEFINED PROFILE VALUES TO FIT NEW GRID
C
      DO 5 K=J+1,I-1
          A(K)=A(J)+FLOAT(K-J)*(A(I)-A(J))/FLOAT(I-J)
    5 CONTINUE
C
C     RESET J VALUE TO THE LAST DEFINED POINT INTERPOLATED TO
C
    6 J=I
      IF(ILAST.GE.0) THEN
     IF(J.LT.ILAST)GO TO 4
      ELSE
          IF(J.LT.M+2)GO TO 4
      ENDIF
C
C     WRITE COMPUTED ARRAY INTO OUTPUT ARRAY
C          OUTPUT ARRAY MUST HAVE NO MORE THAN M VALUES:
C              MUST BE SMALLER THAN WORKING ARRAY
C
      DO K=1,M
     AF(K)=A(K)
      ENDDO
      RETURN
      END
```

C. MODAL EXTRACTION PROGRAM

This program, written in Fortran F77, takes the binary output of the modal decomposition program, and puts it into a usable format for modal beamforming. The output from this program is in two files. The first is an ASCII file

containing the eigen values with corresponding frequencies. The second contains the eigen function, or mode shapes for each frequency stored in a direct access binary format.

```
************************************************************************
*    PROGRAM: "CWextract.f"
*
*       CREATED BY GLENN A. OMANS II
*       WRITTEN 30 JUN 1992
*    LAST UPDATE 18 JULY 1992
*
*    PURPOSE: TO EXTRACT MODE DATA FOR USE IN BB BEAMFORMING
*
*    INPUT: DIRECT ACCESS BINARY DATA CREATED BY ALLFREQ.F
*    FILENAME IS USUALLY modes.dabin
*
*    OUTPUT: TWO FILES CONTAINING THE EIGEN FUNCTION (Z) FOR
*    ALL FREQUENCIES AND THE EIGEN VALUES Kn's
*
*    EIGEN FUNCTION OUTPUT IS A SEQUENTIAL BINARY FILE
*       EIGEN VALUE OUTPUT IS AN ASCII FILE
*
*
************************************************************************
*    LISTING OF VARIABLES, ARRAYS & PARAMETERS
*
* buff      (array) vector holding input eigen functions for a specific
*           frequency
* df        (var)   frequency increment
* dz        (var)   depth increment
* efun      (array) depth by freq. matrix holding eigen function vectors
* eval      (array) eigen value vector for all frequencies & some mode #
* EX        (log)   switch defining a files existance
* fc        (var)   carrier/center frequency, Not used
* fmax      (var)   maximum frequency in band
* fmin      (var)   minimum frequency in band
* fname1  (char)  Holds binary input filename
* fname2  (char)  Holds eigen function filename
* fname3  (char)  Holds eigen value filename
* ifreq   (var)   frequency counter
* irec      (var)   Points to record being read from
* itrap     (var)   Total number of trapped modes for a single frequency
* izmax    (var)   Maximum Number of Depth Points; the min(zmax/dz and
nzp1-1)
* jump      (var)   from input file, defines number of modes stored
* kn        (var)   SQRT(Eigenvalue) for specific mode & frequency
* m    (par)   Maximum Number of eigen values ie. modes
*                  [Must be larger then stored modes]
* mi        (var)   Total number of solutions evaluated by eigen solver
* mintrap (par)   Defines Maximinimum for trapped/saved modes
* mode      (var)   Input from STD IO defining mode # to extract
* n    (par)   Maximum Eigen Function Length
*                  [Must be larger then stored frequencies]
* nf        (var)   Total number of frequencies in file
* Nfc       (var)   Defines the "ifreq" for the Center Frequency
* pi        (var)   PI itself in double precision
* rcl       (var)   Record Length, defined by input file
```

```
* w     (var)   Radial Frequency, part of eigen value output
* z     (array) Depth at each frequency
* zmax      (par)   depth cutoff for eigen function


      implicit real*8 (a-h,o-z)
      parameter(zmax=420)
      parameter(n=210,m=50)
      character*20 fname1,fname2,fname3

      real*8 w,fc,dz,df,kn,efun,z
      real*8 pi
      real*4 fmin,fmax
      integer jump,rcl,irec,izmax
      LOGICAL EX
      DIMENSION eval(m),efun(n+1,m),buff(n)


C
C     DECLARE PI IN DOUBLE PRECISION
C
      pi = 4.d0 * datan(1.d0)
C
C     PRINT HEADER
C
      write(*,*)'\n\n\n'
      write(*,*)'\n\t\t\t\t\t\t\tPROGRAM WRITTEN: 3 AUGUST 1992'
      write(*,*)'\t\t\t\t\t\t\t          BY: Glenn A. Omans II'
      write(*,*)'\n\n'
      write(*,*)'\t*** EXTRACT EIGEN FUNCTIONS AND ',
     &          'VALUES FOR BB BEAMFORMING ***'
      write(*,*)'\n\nNOTE:'
      write(*,*)'INPUT FILE IS DIRECT ACCESS BINARY',
     &          '\n\tUSUALLY CALLED: modes.dabin'
      write(*,*)'\t\tCREATED FROM MODAL DECOMPOSTION PROGRAM'
      write(*,*)'\t\t\tCALLED allfreq.f'
      write(*,*)'\n\n\n'
C
C     READ MODE NUMBER FROM OPERATOR
C

      write(*,*)'\n INPUT MODE NUMBER TO BE EXTRACTED \n'
      read(*,*) mode
C
C     READ IN FILE NAMES FROM KEYBOARD
C

      write(*,*)'\nENTER NAME OF BINARY INPUT FILE:\n'
      read(*,*) fname1
      INQUIRE(FILE=fname1,EXIST=EX)
      IF(.NOT.EX) GOTO 2001
      write(*,*)'\nENTER NAME OF EIGEN FUNCTION OUTPUT FILE:\n'
      read(*,*) fname2
      write(*,*)'\nENTER NAME OF EIGEN VALUE OUTPUT FILE:\n'
      read(*,*) fname3

C
C     OPEN INPUT AND OUTPUT FILES WITH ERROR MSG
C
      OPEN(UNIT=20,FILE=fname1,STATUS='OLD',FORM='UNFORMATED',
     &     ACCESS='DIRECT',RECL=1700,ERR=2000)
```

```
C    ***  ABOVE FILE WILL BE CLOSED & REOPENED ONCE CORRECT  ***
C    ***           RECORD LENGTH HAS BEEN OBTAINED            ***

      INQUIRE(FILE=fname2,EXIST=EX)
      IF(EX) GOTO 2005

C     OPEN EIGEN VALUE DATA FILE AFTER RECORD SIZE DETERMINED

      OPEN(UNIT=21,STATUS='NEW',FORM='FORMATED',FILE=fname3,
     &     ERR=2010)

C
C     READ IN DATA FROM INPUT FILE
C

      read(UNIT=20,REC=1) fc,fmin,fmax,df,nf,dz,mi,nzp1,jump,rcl

      write(*,*)'RECORD SIZE:',rcl
      write(*,*)'fc =',fc
      write(*,*)'fmin =',fmin,' fmax =',fmax
      write(*,*)'df =',df,' nf =',nf
      write(*,*)'dz =',dz,' Max Mode Eval =',mi,' Eigen Size =',nzp1
C
C     CLOSE DATA FILE & REOPEN w/ CORRECT RECORD LENGTH
C
      CLOSE(20)
      OPEN(UNIT=20,FILE=fname1,STATUS='OLD',FORM='UNFORMATED',
     &     ACCESS='DIRECT',RECL=rcl,ERR=2000)


C     ENSURE SELECTED MODE NUMBER IS IN BOUNDS

      if(jump.LT.mi)then
         mintrap=jump
      else mintrap=mi
      endif
C     ERROR OUT ON NOT ENOUGH MODES STORED

      if(mode.GT.mintrap) GOTO 2020

C     ERROR OUT IF VARIABLES ARE DIMENSIONED TOO SMALL

      if(m.LT.nf) then
         GOTO 2030
      endif
      if(n.LT.(nzp1-1)) then
         GOTO 2040
      endif

C
C     EXTRACT EIGEN VALUES
C

      do 100 ifreq=1,nf

         irec=(jump+2)*(ifreq-1) + 4
         read(UNIT=20,REC=irec) w,itrap,eval

C     ERROR OUT ON "MODE NOT STORED"

         itrap=itrap+1
```

```
         if(mintrap.GT.itrap) mintrap=itrap
         if(mode.GT.itrap) goto 2020

C    OUTPUT EIGEN VALUE TO FILE

         kn = sqrt(eval(mode))
         write(21,*) w,kn

 100     CONTINUE

C
C    LOAD EIGEN FUNCTION MATRIX
C

      do 300 ifreq=1,nf
         irec = (jump+2)*(ifreq-1)+4+mode
         read(UNIT=20,REC=irec) buff

         do 200 iz=1,(nzp1-1)
            efun(iz,ifreq) = buff(iz)
 200     continue
 300  continue

C
C    WRITE DEPTH VECTOR TO efun(iz,0)
C

      i=0
      z=0
      do 350 iz=1,(nzp1-1)
         if(z.LE.zmax) then
            i=i+1
            efun(iz,0)=z
         endif
  350 z = z+dz

      izmax=i

C
C    OPEN EIGEN FUNCTION DATA FILE
C

      OPEN(UNIT=22,FILE=fname2,ACCESS='DIRECT',RECL=4*(nf+1),
     &     FORM='UNFORMATED',STATUS='NEW',ERR=2005)


C
C    WRITE EIGEN FUNCTION MATRIX TO BINARY FILE
C

      CALL efunOUT(efun,temp,n,m,nf,izmax)


      GOTO 2050

C
C    ERROR MESSAGES
C

2000  write(*,*)'\n ERROR OPENING INPUT FILE: ',fname1
      write(*,*)'\n INPUT FILE MUST BE BINARY'
```

```
      write(*,*)'\n CHECK RECORD SIZE'
      GOTO 2050
2001  write(*,*)'\n ERROR INPUT FILE ',fname1,' DOES NOT EXIST!!'
      GOTO 2050
2005  write(*,*)'\n ERROR OPENING EIGEN VALUE DATA FILE: ',fname2
      write(*,*)'\n FILE MAY ALREADY EXIST'
      GOTO 2050
2010  write(*,*)'\n ERROR OPENING EIGEN FUNCTION DATA FILE: ',fname3
      write(*,*)'\n FILE MAY ALREADY EXIST'
      GOTO 2050
2020  write(*,*)'\n ERROR ERROR ERROR ERROR ERROR\n'
      write(*,*)'NUMBER OF MODES SELECTED',mode,
     &     ' EXCEEDS STORED MODES',mintrap
      GOTO 2045
2030  write(*,*)'\n ERROR ERROR ERROR ERROR ERROR\n'
      write(*,*)'DIMENSION ERROR, m =',m
      write(*,*)'MUST BE GREATER THEN',jump
      GOTO 2045
2040  write(*,*)'\n ERROR ERROR ERROR ERROR ERROR\n'
      write(*,*)'DIMENSION ERROR, n =',n
      write(*,*)'MUST BE GREATER THEN',nf
2045  close(21,STATUS='DELETE')
      close(22,STATUS='DELETE')
      GOTO 2060

2050  close(21)
      close(22)
2060  close(20)

      STOP
      END
*****************************************************************
*****************************************************************
* SUBROUTINE efunOUT(efun,temp,n,m,nf,izmax)
*
* WRITE TO DIRECT ACCESS FILE
*
*****************************************************************
      SUBROUTINE efunOUT(efun,temp,n,m,nf,izmax)

      real*8 efun
      real*4 temp
      integer iz,n,m,nf,izmax
      dimension efun(n+1,m),temp(nf+1)
C
C    INITIALIZE TEMP BUFFER
C

      do ifreq=1,nf
         temp(iz)=0.0
      enddo

      do 500 iz=1,izmax
         do 400 ifreq=0,nf
            temp(ifreq+1) = float(efun(iz,ifreq))
  400    continue
         write(22,rec=iz) temp
  500 continue

      RETURN
      END
```

D. MODAL DECOMPOSITION SAMPLE INPUT FILE

```
395.0d0                          depth (m)
2.0d0                             dz (m)
224.d0                           Center Freq (Hz)
  0.0000000e+00   1.4790844e+03
  8.0000000e+00   1.4798710e+03
  1.0000000e+01   1.4798281e+03
  1.2000000e+01   1.4797952e+03
  1.4000000e+01   1.4797784e+03
  1.6000000e+01   1.4797665e+03
  1.8000000e+01   1.4798083e+03
  2.0000000e+01   1.4798116e+03
  2.2000000e+01   1.4796737e+03
  2.4000000e+01   1.4787948e+03
  2.6000000e+01   1.4752782e+03
  2.8000000e+01   1.4738331e+03
  3.0000000e+01   1.4731201e+03
  3.2000000e+01   1.4714027e+03
  3.4000000e+01   1.4711126e+03
  3.6000000e+01   1.4703097e+03
  3.8000000e+01   1.4698482e+03
  4.0000000e+01   1.4690343e+03
  4.2000000e+01   1.4687429e+03
  4.4000000e+01   1.4687005e+03
  4.6000000e+01   1.4684512e+03
  4.8000000e+01   1.4682411e+03
  5.0000000e+01   1.4681399e+03
  5.2000000e+01   1.4679103e+03
  5.4000000e+01   1.4678015e+03
  5.6000000e+01   1.4676039e+03
  5.8000000e+01   1.4673772e+03
  6.0000000e+01   1.4672268e+03
  6.2000000e+01   1.4671120e+03
  6.4000000e+01   1.4669098e+03
  6.6000000e+01   1.4668384e+03
  6.8000000e+01   1.4666859e+03
  7.0000000e+01   1.4665955e+03
  7.2000000e+01   1.4665304e+03
  7.4000000e+01   1.4665222e+03
  7.6000000e+01   1.4664575e+03
  7.8000000e+01   1.4663630e+03
  8.0000000e+01   1.4662527e+03
  8.2000000e+01   1.4661536e+03
  8.4000000e+01   1.4661243e+03
  8.6000000e+01   1.4661243e+03
  8.8000000e+01   1.4661387e+03
  9.0000000e+01   1.4661316e+03
  9.2000000e+01   1.4661271e+03
  9.4000000e+01   1.4657917e+03
  9.6000000e+01   1.4657545e+03
  9.8000000e+01   1.4656872e+03
  1.0000000e+02   1.4656550e+03
  1.0200000e+02   1.4656384e+03
  1.0400000e+02   1.4655749e+03
  1.0600000e+02   1.4655615e+03
  1.0800000e+02   1.4655545e+03
  1.1000000e+02   1.4655444e+03
  1.1200000e+02   1.4654871e+03
  1.1400000e+02   1.4654259e+03
```

1.1600000e+02	1.4652586e+03
1.1800000e+02	1.4651952e+03
1.2000000e+02	1.4651795e+03
1.2200000e+02	1.4650894e+03
1.2400000e+02	1.4650565e+03
1.2600000e+02	1.4649848e+03
1.2800000e+02	1.4648727e+03
1.3000000e+02	1.4648223e+03
1.3200000e+02	1.4647187e+03
1.3400000e+02	1.4646833e+03
1.3600000e+02	1.4646635e+03
1.3800000e+02	1.4645808e+03
1.4000000e+02	1.4645149e+03
1.4200000e+02	1.4644538e+03
1.4400000e+02	1.4642631e+03
1.4600000e+02	1.4642143e+03
1.4800000e+02	1.4641319e+03
1.5000000e+02	1.4640558e+03
1.5200000e+02	1.4639944e+03
1.5400000e+02	1.4638786e+03
1.5600000e+02	1.4638155e+03
1.5800000e+02	1.4637510e+03
1.6000000e+02	1.4636782e+03
1.6200000e+02	1.4636094e+03
1.6400000e+02	1.4635256e+03
1.6600000e+02	1.4633304e+03
1.6800000e+02	1.4632558e+03
1.7000000e+02	1.4632098e+03
1.7200000e+02	1.4631657e+03
1.7400000e+02	1.4629990e+03
1.7600000e+02	1.4629380e+03
1.7800000e+02	1.4629091e+03
1.8000000e+02	1.4628036e+03
1.8200000e+02	1.4627138e+03
1.8400000e+02	1.4626069e+03
1.8600000e+02	1.4625955e+03
1.8800000e+02	1.4625605e+03
1.9000000e+02	1.4624790e+03
1.9200000e+02	1.4624430e+03
1.9400000e+02	1.4624121e+03
1.9600000e+02	1.4623541e+03
1.9800000e+02	1.4622126e+03
2.0000000e+02	1.4621396e+03
2.0200000e+02	1.4620576e+03
2.0400000e+02	1.4619457e+03
2.0600000e+02	1.4618913e+03
2.0800000e+02	1.4618832e+03
2.1000000e+02	1.4618745e+03
2.1200000e+02	1.4617301e+03
2.1400000e+02	1.4616586e+03
2.1600000e+02	1.4615538e+03
2.1800000e+02	1.4615089e+03
2.2000000e+02	1.4614916e+03
2.2200000e+02	1.4614802e+03
2.2400000e+02	1.4614468e+03
2.2600000e+02	1.4613521e+03
2.2800000e+02	1.4613124e+03
2.3000000e+02	1.4611986e+03
2.3200000e+02	1.4611175e+03
2.3400000e+02	1.4609844e+03
2.3600000e+02	1.4607806e+03

```
  2.3800000e+02   1.4606755e+03
  2.4000000e+02   1.4605276e+03
  2.4200000e+02   1.4598254e+03
  2.4400000e+02   1.4594292e+03
  2.4600000e+02   1.4591842e+03
  2.4800000e+02   1.4590498e+03
  2.5000000e+02   1.4587578e+03
  2.5200000e+02   1.4587161e+03
  2.5400000e+02   1.4587252e+03
  2.5600000e+02   1.4587113e+03
  2.5800000e+02   1.4587202e+03
  2.6000000e+02   1.4587130e+03
  2.6200000e+02   1.4586958e+03
  2.6400000e+02   1.4586952e+03
  2.6600000e+02   1.4586926e+03
  2.6800000e+02   1.4583988e+03
  2.7000000e+02   1.4572621e+03
  2.7200000e+02   1.4572621e+03
  2.7400000e+02   1.4572621e+03
276.     1677.0
278.     1677.0
280.     1677.0
326.     1677.0
328.     1677.0
395.     1677.0
-1,-1
    0.        1.0              z (m)   rho(cm/g^3)
  272.0       1.0
  274.0       1.0
  276.0       1.83
  278.0       1.83
  300.0       1.83
  302.0         1.83
  384.0       1.83
  395.0       1.83
-1,-1
```

REFERENCES

1. Munk, W. and Wunsch, C., "Ocean Acoustic Tomography: A scheme for Large Scale Monitoring," Deep-Sea Research, v. 26A pp. 123-161, 1979.

2. Shang, E.C., "Ocean Acoustic Tomography based on Adiabatic Mode Theory," J. Acoust. Soc. Am., 85 (4), pp. 1531-1537, 1989.

3. Munk, W. and Wunsch, C., "Ocean Acoustic Tomography: Rays and Modes," Reviews of Geophysics and Space Physics, Vol 21, No. 4, pp. 777-793, 1983.

4. Crocker, S.E., *Time Domain Modal Beamforming for a near Vertical Acoustic Array*, MS Thesis, Naval Postgraduate School, Montery, CA, December 1991.

5. Westreich, E.L., *Modeling Pulse Transmissions in the Monterey Bay using Parabolic Equation Methods*, MS Thesis, Naval Postgraduate School, Monterey, CA, September 1991.

6. Elliott, J.M., *Simulation of Acoustic Multipath Arrival Structure in the Barents Sea*, MS Thesis, Naval Postgraduate School, Monterey, CA, June 1992.

7. Bourke, R., Chiu, C., Lynch, J., Miller, J., Muench, R., Plueddemann, A., "Preliminary Cruise Results Barents Sea Polar Front Experiment," Barents Sea Polar Front Group, Unpublished, August 1992.

8. Kinsler, L.E. and others, *Fundementals of Acoustics*, 3rd edition, John Whiley & Sons, 1982.

9. Burdic, W.S., *Underwater Acoustics System Analysis*, Prentice-Hall, 1984.

10. Ziomek, L.J., *Underwater Acoustics: A Linear Systems Theory Approach*, Academic Press Inc., 1985.

11. Eldred, R.M., *Doppler Processing of Phase Encoded Underwater Acoustic Signals*, MS Thesis, Naval Post Graduate School, Monterey, CA, September 1990.

12. Ziemer, R. and Peterson, R., *Digital Communications and Spread Spectrum Systems*, Macmillan Publishing Company, 1985.

13. Chiu, C.S., Ehret, L.L. and Westreich, E.L., Department of Oceanography, Naval Postgraduate School, personal communication, 1992.

14. Clay, C.S. and Medwin, H., *Acoustical Oceanography*, John Whiley Sons Inc., 1977.

15. Urick, R.J., *Principles of Underwater Sound*, 3rd edition, McGraw-Hill Inc., 1983.

16. O'Neal, P.V., *Advanced Engineering Mathmatics*, 2nd edition, Wadsworth Inc., 1987.

17. Press, W.H. and others, *Numerical Recipies in C*, Cambridge University Press, 1988.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6151	2
2.	Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Prof. J.H. Miller, Code EC/Mr Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943	4
4.	LCDR P.J. Rovero, Code OC/Rv Department of Oceanography Naval Postgraduate School Monterey, CA 93943	1
5.	Prof. C.S. Chiu OC/Ci Department of Oceanography Naval Postgraduate School Monterey, CA 93943	1
6.	LT G.A. Omans II Box LW-19 Felton, DE 19943	1
7.	Dr. J.F. Lynch Woods Hole Oceanographic Institution Woods Hole, MA 02543	1
8.	Mr. K. Von der Heydt Woods Hole Oceanographic Institution Woods Hole, MA 02543	1
9.	LCDR T. Bily CINCLANTFLT Code N351A Norfolk, VA 23511-5105	1
10.	Dr. W. Jobst NAVOCEANO Code 0 Stennis Space Center, MS 39522	1